Parallel Order-Based Core Maintenance in Dynamic Graphs

Abstract

The core numbers of vertices in a graph are one of the most well-studied cohesive subgraph models because of the linear running time. In practice, many data graphs are dynamic graphs that are continuously changing by inserting or removing edges. The core numbers are updated in dynamic graphs with edge insertions and deletions, which is called core maintenance. When a burst of a large number of inserted or removed edges come in, we have to handle these edges on time to keep up with the data stream. There are two main sequential algorithms for core maintenance, Traversal and Order. The experiments show that the Order algorithm significantly outperforms the Traversal algorithm over a variety of real graphs.

To the best of our knowledge, all existing parallel approaches are based on the Traversal algorithm. These algorithms exploit parallelism only for vertices with different core numbers; they reduce to sequential algorithms when all vertices have the same core numbers. In this paper, we propose a new parallel core maintenance algorithm based on the Order algorithm. More importantly, our new approach always has parallelism, even for graphs where all vertices have the same core numbers. Extensive experiments are conducted over real-world, temporal, and synthetic graphs on a multicore machine. The results show that for inserting and removing a batch of edges using 16 workers, our method achieves up to 289x and 10x times speedups compared with the most efficient existing method, respectively.

CCS Concepts

• **Computing methodologies → Shared memory algorithms**.

Keywords

Dynamic Graphs, k-Core Maintenance, Parallel, Multicore

ACM Reference Format:

. 2023. Parallel Order-Based Core Maintenance in Dynamic Graphs. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Graphs are widely used to model complex networks. As one of the well-studied cohesive subgraph models, the k-core is defined as the maximal subgraph such that all vertex have degrees at least k. Here, the *core number* of a vertex is defined as the maximum value of k such that this vertex is contained in the subgraph of k-core [1, 17]. The core numbers can be computed with linear time $O(m)$ by the BZ algorithm [1], where m is the number of edges in a graph. Due to such computational efficiency, the core number of a vertex can be a parameter of density extensively used in numerous applications [17], such as knowledge discovery [30], gene expression [8], social networks [10], ecology [3], and finance [3].

In [21], Malliaros, et al. summarize the main research work related to k-core decomposition from 1968 to 2019. Many papers focus on computing the core in static graphs [1, 4, 16, 23, 31]. In practice, many data graphs are both large and continuously changing. It is important to identify the dense range as fast as possible after a change, e.g., multiple edges are inserted or removed. For example, it is necessary to quickly initiate a response to rapidly spreading false information about vaccines or to urgently address new pandemic super-spreading events [22, 24, 9]. This is a problem of maintaining the core number in dynamic graphs. In [35], Zhang, et al. summarize the research on core maintenance and applications.

Many sequential algorithms are devised on core maintenance in dynamic graphs [12, 36, 26, 33, 25, 18]. The main idea for core maintenance is that we first need to identify a set of vertices whose core numbers need to be updated (denoted as V^*) by traversing a possibly larger cope of vertices (denoted as V^+). There are two main algorithms, Order [36] and Traversal [26]. Given an inserted edge, the Order algorithm has to traverse much fewer vertices than the Traversal algorithm by maintaining the order for all vertices. That is why the Order algorithm has significantly improved running time. In [12], a Simplified-Order algorithm is proposed for easy understanding and implementation based on the Order algorithm.

All the above methods are sequential for maintaining core numbers over dynamic graphs, which means each time only one insert or removal edge is handled. The problem is that when a burst of a large number of inserted or removed edges come in, these edges may not be handled on time to keep up with the data stream [9]. The prevalence of multi-core machines suggests parallelizing the core maintenance algorithms. Many multi-core parallel batch algorithms for core maintenance have been proposed in [13, 14, 29]. All above methods have similar ideas: 1) they use an available structure, e.g. *Join Edge Set* [13] or *Matching Edge Set* [14], to preprocess a batch of inserted or removed edges avoiding repeated computations, and 2) each worker performs the Traversal algorithm. There are two drawbacks to these approaches. First, they are based on the sequential Traversal algorithm [25, 18], which is much less efficient than the Order algorithm [13, 12]. Second, they exploit the parallelism only for vertices with different core numbers, that is, they reduce to sequentially algorithms when all affected vertices have the same core numbers.

To overcome the above drawbacks, inspired by the Simplified-Order algorithm [12], we propose a new parallel algorithm to maintain core numbers after for dynamic graphs, so-called the Parallel-Order algorithm. That is, each worker handles one inserted or removed edge at a time and propagates the affected vertices in order, and we lock vertices for synchronization. The parallel order maintenance data structure [11] is adopted to maintain the order for all vertices. We use the work and depth model to analyze our parallel algorithm, where the work is its sequential running time,

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

and the depth is its running time on an infinite number of processors [5]. For edge insertion and removal, our parallel approach has the same work as the sequential Simplified-Order algorithm [12]. The main contributions of our work are summarized below:

- For edge insertion and removal, we design novel mechanisms for synchronization by only locking vertices in V^+ instead of locking all accessed edges. In other words, all the neighbors of vertices in V^+ are not necessarily locked. This is meaningful considering real graphs always have a much larger number of edges than vertices, let alone dense graphs. Additionally, for each inserted or removed edge, the size of V^+ is typically less than 10. Thus, it has a low probability that multiple workers block as a chain and then reduce to sequential execution. Fewer locked vertices will lead to higher parallelism.
- For edge insertion, we lock all vertices in order to avoid deadlocks. For edge removal, we design a conditional lock mechanism to avoid deadlocks. We prove that deadlocks will never happen.
- We conduct extensive experiments on a multicore machine over various graphs. Our method achieves significant up to 5x times speedups by using 16 workers for edge insertion and removal. Compared with the existing most efficient parallel approach in [13], our method achieves up to 289x and 10x times speedups for edge insertion and removal, respectively.

The rest of this paper is organized as follows. The related work is discussed in Section 2. The preliminaries are given in Section 3. Our new parallel Order-Based core maintenance algorithms are proposed in Section 4. We conduct extensive performance studies and show the results in Section 5, and conclude in Section 6.

2 Related Work

Core Decomposition. The BZ algorithm [1] has linear running time $O(m)$ by using a bucket structure, where m is the number of edges. In [4], an external memory algorithm is proposed, so-called EMcore, which runs in a top-down manner such that the whole graph does not have to be loaded into memory. In [31], Wen et al. provide a semi-external algorithm, which requires $O(n)$ size memory to maintain the information of vertices, where n is the number of vertices. In [16], Khaouid et al. investigate the core decomposition in a single PC over large graphs by using `GraphChi` and `WebGraph` models. In [23], Montresoret et al. consider the core decomposition in a distributed system. In addition, the parallel computation of core decomposition in multi-core processors is first investigated in [6], where the ParK algorithm was proposed. Based on the main idea of ParK, a more scalable PKC algorithm has been reported in [15].

Core Maintenance. In [25, 18], an algorithm that is similar to the Traversal algorithm is given, but this solution has quadratic time complexity. In [31], a semi-external algorithm for core maintenance is proposed in order to reduce the I/O cost, but this method is not optimized for CUP time. In [27], Sun et al. design algorithms to maintain approximate cores in dynamic *hypergraphs* in which a *hyteredge* may contain one or more participating vertices compared with exactly two in graphs. In [9], Gabert et al. propose parallel core maintenance algorithms for maintaining cores over hypergraphs. There exists some research based on core maintenance. In [34], the authors study computing all k-cores in the graph snapshot over the time window. In [19], the authors explore the hierarchy core maintenance. In [32], the distributed approaches to core maintenance are explored.

Weighted Graphs. All the above work focus on unweighted graphs, but graphs are weighted in a lot of realistic applications. For an edge-weighted graph, the degree of a vertex is the sum of the weights of all its incident edges. But it has a large search range to maintain the core numbers after the change by using the traditional core maintenance algorithms directly, as the degree of a related vertex may change widely. In [37], Zhou et al. extend the coreness to weighted graphs and devise weighted core decomposition algorithms; also they devise weighted core maintenance based on the k-order [36, 12]. In [20], Liu et al. improve the core decomposition and incremental maintenance algorithm to suit edge-weighted graphs.

3 Preliminaries

Let $G = (V, E)$ be an undirected unweighted graph, where $V(G)$ denotes the set of vertices and $E(G)$ represents the set of edges in G. When the context is clear, we will use V and E instead of $V(G)$ and $E(G)$ for simplicity, respectively. As G is an undirected graph, an edge $(u, v) \in E(G)$ is equivalent to $(v, u) \in E(G)$. We denote the number of vertices and edges of G by n and m, respectively. The set of neighbors of a vertex $u \in V$ is defined by $u.adj = \{v \in V : (u, v) \in E\}$. The degree of a vertex $u \in V$ is defined by $u.deg = |u.adj|$.

Definition 3.1 (k-Core)**.** Given an undirected graph $G = (V, E)$ and a natural number k, a subgraph G_k of G is called a k-core if it satisfies: (1) for $\forall u \in V(G_k)$, $u.deg \geq k$, and (2) G_k is maximal. Moreover, $G_{k+1} \subseteq G_k$, for all $k \geq 0$, and G_0 is just G.

Definition 3.2 (Core Number)**.** Given an undirected graph $G = (V, E)$, the core number of a vertex $u \in G(V)$, denoted as $u.core$, is defined as $u.core = max\{k : u \in V(G_k)\}$. That means $u.core$ is the largest k such that there exists a k-core containing u.

Definition 3.3 (k-Subcore)**.** Given a undirected graph $G = (V, E)$, a maximal set of vertices $S \subseteq V$ is called a k-subcore if (1) $\forall u \in S, u.core = k$; (2) the induced subgraph $G(S)$ is connected. The subcore that contain vertex u is denoted as $\mathtt{sc}(u)$.

Core Decomposition. Given a graph G, the problem of computing the core number for each $u \in V(G)$ is called core decomposition. In [1], Batagelj et al. propose a liner time $O(m + n)$ algorithm, so-called BZ algorithm. The general idea is peeling: To compute the k-core G_k of G, it repeatedly removes those vertices (and their adjacent edges) whose degrees are less than k. When there are no more vertices to remove, the resulting graphs are the k-core of G. In this algorithm, the min-priority queue Q can be efficiently implemented by bucket sorting [1] which leads to a linear running time of $O(m + n)$.

Core Maintenance. The core numbers for dynamic graphs G should be maintained when edges are inserted into and removed from G continuously. The insertion and removal of vertices can be simulated as a sequence of edge insertions and removals.

Definition 3.4 (Candidate Set V^* and Searching Set V^+)**.** Given a graph $G = (V, E)$, when an edge is inserted or removed, a candidate set of vertices, denoted as V^*, needs to be identified and the core numbers of vertices in V^* must be updated. To identify V^*, a

minimal set of vertices, denoted as V^+, is traversed by accessing their adjacent edges.

Clearly, we have $V^* \subseteq V^+$ and an efficient core maintenance algorithm should have a small ratio of $|V^+|/|V^*|$. The Order [13] insertion algorithm has a significantly smaller such ratio compared with the Traversal [25] insertion algorithm. This is why we try to parallelize the Order algorithm in this paper.

In [18, 25], it is proved that after inserting or removing one edge, the core number of vertices in V^* increase or decrease at most one, respectively; the V^* only located in the k-subcore, where k is the lower core number of two vertices that the inserted or removed edge connect.

3.1 The Order-Based Core Maintenance

The state-of-the-art core maintenance solution is the Order algorithm [36, 12]. For edge insertion, it is based on three notions, namely k-order, candidate degree, and remaining degree. For edge removal, it uses the notion of a max-core degree [25].

Edge Insertion.

Definition 3.5 (k-Order $\preceq$)**.** [36] Given a graph G, the k-order $\preceq$ is defined for any pairs of vertices u and v over the graph G as follows: (1) when $u.core < v.core$, then $u \preceq v$; (2) when $u.core = v.core$, then $u \preceq v$ if u's core number is determined before v's by the peeling steps of BZ algorithm.

A k-order $\preceq$ is an instance of all the possible vertex sequences produced by the BZ algorithm. The k-order is transitive, that is, $u \preceq v$ if $u \preceq w \wedge w \preceq v$. For each edge insertion and removal, the k-order has to be maintained. Here, $\mathbb{O}_k$ denotes the sequence of vertices in k-order whose core numbers are k. A sequence $\mathbb{O} = \mathbb{O}_0\mathbb{O}_1\mathbb{O}_2\cdots$ over $V(G)$ can be obtained, where $\mathbb{O}_i \preceq \mathbb{O}_j$ if $i < j$. It is clear that $\preceq$ is defined over the sequence of $\mathbb{O} = \mathbb{O}_0\mathbb{O}_1\mathbb{O}_2\cdots$. In other words, for all vertices in a graph, the sequence $\mathbb{O}$ indicates the k-order $\preceq$.

Given an undirected graph $G = (V, E)$ with $\mathbb{O}$ in k-order, each edge $(u, v) \in E(G)$ can be assigned a direction such that $u \preceq v$. By doing this, a *direct acyclic graph* (DAG) $\vec{G} = (V, \vec{E})$ can be constructed where each edge $u \mapsto v \in \vec{E}(\vec{G})$ satisfies $u \preceq v$. Of course, the k-order of G is a topological order of $\vec{G}$. Here, the successors of v is defined as $u(\vec{G}).post = \{v \mid u \mapsto v \in \vec{E}(\vec{G})\}$; the predecessors of v is defined as $u(\vec{G}).pre = \{v \mid v \mapsto u \in \vec{E}(\vec{G})\}$. When the context is clear, we use $u.post$ and $u.pre$ instead of $u(\vec{G}).post$ and $u(\vec{G}).pre$, respectively [12].

Definition 3.6 (candidate in-degree)**.** [12, 36] Given a constructed DAG $\vec{G}(V, \vec{E})$, the candidate in-degree $v.d^*_{in}$ is the total number of v's predecessors located in V^*, denoted as $d^*_{in}(v) = |\{w \in v.pre : w \in V^*\}|$.

Definition 3.7 (remaining out-degree)**.** [12, 36] Given a constructed DAG $\vec{G}(V, \vec{E})$, the remaining out-degree $v.d^+_{out}$ is the total number of v's successors without the ones that are confirmed not in V^*, denoted as $v.d^+_{out} = |\{w \in v.post : w \notin V^+ \setminus V^*\}|$.

For all vertices v in $\vec{G}$, we must ensure that $v.core \leq v.d^+_{out}$. When inserting an edge $v \mapsto u$, we have $v.d^+_{out}$ increased by 1. If $v.core > v.d^+_{out}$, edge insertion maintenance is required.

The Order insertion algorithm [12] is presented in Algorithm 1. Before the algorithm, we assume that all vertices v have correctly initialized $v.core$, $v.d^+_{out}$, and $v.d^*_{in}$. After inserting an edge $u \mapsto v$, we add u to V^* when $u.d^+_{out} > K = u.core$ (line 2). The key idea is that we repeatedly add vertices w to V^* such that $w.d^*_{in}+w.d^+_{out} > K$. Importantly, the priority queue Q is used to traverse all affected vertices in k-order (lines 4 - 6). When traversing the affected vertices w in $\mathbb{O}$, the value $w.d^*_{in}+w.d^+_{out}$ is the upper-bound as we traverse $\vec{G}$ in topological order. There are two cases. First, if $w.d^*_{in}+w.d^+_{out} > K$, we execute the `Forward` procedure to add w into V^*; also, for each $w' \in w.post$ with $w'.core = K$, we add $w'.d^*_{in}$ by 1 and then add to Q for propagation (line 7). Second, if $w.d^*_{in}+w.d^+_{out} \leq K \wedge w.d^*_{in} > 0$, we identify that w cannot possibly be added in V^*; the `Backward` procedure propagates w to remove potential vertices v from V^* since $v.d^+_{out}$ or $v.d^*_{in}$ is decreased (line 8). All other vertices w in $\mathbb{O}$ that are not in the above two cases are skipped. Finally, V^* includes all vertices whose core numbers should add by 1 (line 9). Of course, the k-order is maintained for inserting other edges (line 10).

Algorithm 1: InsertEdge($\vec{G}, \mathbb{O}, u \mapsto v$)

```
1  V*, V+, K ← ∅, ∅, u.core
2  insert u ↦ v into G⃗ with u.d+_out ← u.d+_out + 1
3  if u.d+_out ≤ K then return
4  Q ← a min-priority queue by 𝕆; Q.enqueue(u)
5  while Q ≠ ∅ do
6      w ← Q.dequeue()
7      if w.d*_in + w.d+_out > K then Forward(w, V*, V+)
8      else if w.d*_in > 0 then Backward(w, V*, V+)
9  for w ∈ V* do w.core ← K + 1; w.d*_in ← 0
10 To maintain the k-order, remove each w ∈ V* from 𝕆_K and insert
   w at the beginning of 𝕆_{K+1} in k-order.
```

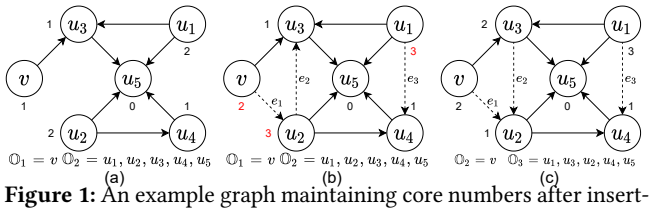

Figure 1: An example graph maintaining core numbers after inserting edges, e_1, e_2, and e_3. The letters inside the cycles are vertices' IDs and order $\mathbb{O}_k$ is the k-order of vertices with core numbers k. The number beside each vertex is its remaining out-degree d^+_{out}. The direction for each edge indicates the k-order of two vertices, which is constructed as a DAG. (a) the initial graph. (b) after inserting 3 edges. (c) the core numbers and k-orders are updated.

Example 3.1. In Figure 1, we show an example graph that maintains the core numbers of vertices after inserting edges, e_1 to e_3, successively. Figure 1(a) shows an example graph constructed as a DAG where the direction of edges indicates the k-order. After initialization, v has a core number 1 with k-order $\mathbb{O}_1$ and u_1 to u_5 have a core number 2 with k-order $\mathbb{O}_2$.

Figure 1(b) shows edges, e_1 to e_3, are inserted. (1) For e_1, we increase $v.d^+_{out}$ to 2 so that $v.d^+_{out} > v.core$ and $V^* = \{v\}$. Then, we stop since all $v.post$ have core numbers larger than $v.core$. Finally, we increase $v.core$ from 1 to 2. (2) For e_2, we increase $v.d^+_{out}$ to 3 so that $v.d^+_{out} > v.core$ and $V^* = \{u_2\}$. Then, we traverse u_3 in k-order and find that $u_3.d^*_{in} + u_3.d^+_{out} = 1 + 1 = 2 \leq K = 2$, so that u_3 cannot add to V^* which cause u_2 to be removed from V^* (by `Backward`). In this case, u_2 is moved after u_1 in the k-order as $\mathbb{O}_2 = u_1, u_3, u_2, u_4, u_5$. (3) For e_3, we increase $u_1.d^+_{out}$ to 3 so that $u_1.d^+_{out} > K = 2$ and $V^* = \{u_2\}$. Then, we traverse u_3, u_2, u_4 and u_5 in k-order, all of which are added into V^* since their $d^*_{in} + d^+_{out} > K = 2$. Finally, we increase the core numbers of u_2 to u_5 from 2 to 3.

Figure 1(c) shows the result after inserting edges. We can see all vertices have their core numbers increased by 1. Orders $\mathbb{O}_2$ and $\mathbb{O}_3$ are updated accordingly. All vertices' d^+_{out} are updated accordingly.

Edge Removal.

Definition 3.8 (max-core degree *mcd*)**.** [26, 36, 12] Given a graph $G = (V, E)$, for each vertex $v \in V$, the max-core degree is the number of v's neighbors w such that $w.core \geq v.core$, defined as $v.mcd = |\{w \in v.adj : w.core \geq v.core\}|$.

For all vertices v in G, we have $v.mcd \geq v.core$. When removing an edge (u, v) such that $v.core \geq u.core$, we have $v.mcd$ off by 1. If $v.mcd < v.core$, Edge removal maintenance is required.

The Order removal algorithm is presented in Algorithm 2. After an edge is removed, the affected vertices, u and v, have to be put into V^* if their *mcd* is less than *core* (lines 2 to 4), which may repeatedly cause the *mcd* of other vertices to be decreased and then added to V^* (lines 5 to 9). The queue R is used to propagate the vertices added to V^* whose *mcd* are less than their core numbers (lines 5 and 6). The k-order is maintained for inserting an edge the next time (line 11). Also, all vertices' *mcd* have to be updated for removing an edge the next time (line 12).

Algorithm 2: RemoveEdge($G, \mathbb{O}, (u, v)$)

```
1  R, K, V* ← an empty queue, Min(u.core, v.core), ∅
2  remove (u, v) from G⃗ with updating u.mcd and v.mcd
3  if u.mcd < K then V* ← V* ∪ {u}; R.enqueue(u)
4  if v.mcd < K then V* ← V* ∪ {v}; R.enqueue(v)
5  while R ≠ ∅ do
6      w ← R.dequeue()
7      for w' ∈ w.adj with w'.core = K ∧ w ∉ V* do
8          w'.mcd ← w'.mcd − 1
9          if w'.mcd < K then V* ← V* ∪ {w'}; R.enqueue(w')
10 for w ∈ V* do w.core ← w.core − 1
11 Remove all w ∈ V* from 𝕆_K and append to 𝕆_{K−1} in k-order
12 update mcd for all related vertices accordingly
```

Example 3.2. In Figure 2, we show an example graph that maintains the core numbers of vertices after removing three edges, e_1 to e_3, successively. Figure 2(a) shows that v has a core number of 2 with k-order $\mathbb{O}_2$ and all u_1 to u_5 have core numbers of 3 with k-order $\mathbb{O}_3$. We can see that for all vertices the core numbers are less or equal to *mcd*.

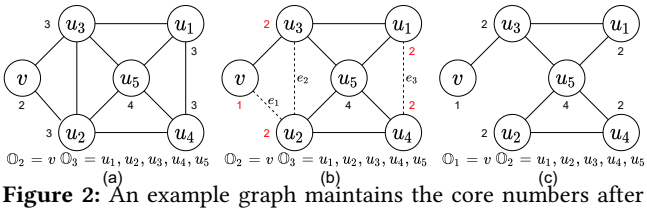

Figure 2: An example graph maintains the core numbers after removing 3 edges, e_1, e_2, and e_3. The letters inside the cycles are vertices' IDs and the $\mathbb{O}_k$ is the k-order of vertices with core numbers k. The beside numbers are corresponding *mcd*. (a) an initial example graph. (b) remove three edges. (c) the core numbers and $\mathbb{O}_k$ update.

Figure 2(b) shows edges, e_1, e_2 and e_3, removed. (1) For e_1, $v.mcd$ is off by 1 so that we have $v.mcd < K = 2$ and $V^* = \{v\}$, but $u_2.mcd$ is not affected. Then, there is no propagation since all $v.adj$ have core numbers larger than $K = 2$. Finally, we decrease $v.core$ from 2 to 1. (2) For e_2, both $u_2.mcd$ and $u_3.mcd$ are off by 1 and less than $K = 3$, so that $V^* = \{u_2, u_3\}$. Then, both u_2 and u_3 are added into R for propagation, and u_1, u_4 and u_5 are consecutively added into V^* with $V^* = \{u_2, u_3, u_1, u_4, u5\}$. Finally, we decrease the core numbers of u_1 to u_5 from 3 to 2; also, the *mcd* of both u_2 and u_3 are updated to 2, and the *mcd* of u_1, u_4 and u_5 are updated to 3. (3) For e_3, both $u_2.mcd$ and $u_3.mcd$ are off by 1. But their *mcd* are still not less than $K = 2$ so that $V^* = \emptyset$. The propagation stop. Finally, the *mcd* of both u_1 and u_4 are updated to 2.

Figure 2(c) shows the result after removing edges. We can see that all vertices have their core numbers decreased by 1. Orders $\mathbb{O}_1$ and $\mathbb{O}_2$ are updated accordingly. Also, all vertices' *mcd* are updated accordingly.

3.2 OM Data Structure

In the Simplified-Order core maintenance algorithm [12], the OM data structure [7, 2] is used to maintain the k-order. In this work, we adopt a concurrent version of the OM data structure [11] to maintain the k-order in parallel. The OM data structure has the following three operations:

- `Order`$(\mathbb{O}, x, y)$: determine if x precedes y in the ordered list $\mathbb{O}$;
- `Insert`$(\mathbb{O}, x, y)$: insert a new item y after x in the ordered list $\mathbb{O}$;
- `Delete`$(\mathbb{O}, x)$: delete x from the total order in the ordered list $\mathbb{O}$.

Assume that there are maximal N items in the total order $\mathbb{O}$. All items are assigned labels to indicate the order. In terms of the `Insert` operation, a *two-level* data structure [28] is used. That is, each item is stored in a bottom-list, which contains a *group* of consecutive elements; each group is stored in a top-list, which can contain $\Omega(\log N)$ items. 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. When there is enough label space after x, y can successfully obtain a new label in $O(1)$ time. Otherwise, the x's group g is *full*, which triggers a *relabel* process. Specifically, the relabel operations have two steps:

- *Rebalance*: if there has no label space after x's group g, we have to rebalance the top-labels of groups. From g, we continuously traverse the successors g' until $L(g') - L(g) > j^2$, where j is the number of traversed groups. Then, new group labels can be assigned with the gap j, in which newly created groups can be inserted. Finally, a new group can be inserted after g.

- *Split*: when x' group g is full, g is split out one new group, which contains at most $\frac{\log N}{2}$ items and new bottom-labels L_b are uniformly assigned for items in new groups. Newly created groups are inserted after g, where we can create the label space by the above rebalance operation.

The `Delete` and `Order` operations cost $O(1)$ time; also, the `insert` operation costs amortized $O(1)$ time. In this work, we adopt the parallel OM data structure [11] to maintain the k-order of all vertices.

3.3 Atomic Primitive and Lock

The compare-and-swap atomic primitive $\texttt{CAS}(x, a, b)$ takes a variable (location) x, an old value a, and a new value b. It checks the value of x, and if it equals a, it updates the variable to b and returns *true*; otherwise, it returns *false* to indicate that updating failed. In this work, we use locks for synchronization in our parallel algorithms. The lock operations can be implemented by `CAS` which is available in most modern architectures. Using the variable x as a lock, the `CAS` will repeatedly check x, and set x from `false` to `true` if x is `false`.

We implement a condition-lock as in Algorithm 5. The condition c is checked before and after the `CAS` lock (lines 1 and 3). It is possible that other workers may update the condition c simultaneously. If c is changed to `false` after locking x, x will be unlocked and then return `false` immediately (line 4). Such a conditional `Lock` can atomically lock x by satisfying c and thus can avoid blocking on a locked x that does not satisfy the condition c.

Algorithm 3: Lock(x) with c

```
1 while c do
2     if x = false ∧ CAS(x, false, true) then
3         if c then return true
4         else x ← false; return false
5 return false
```

4 Parallel Order-Based Core Maintenance

The existing parallel core maintenance algorithms are based on the sequential Traversal algorithm which is experimentally shown much less efficient than the sequential Order algorithm. In this section, based on the Order algorithm, we propose a new parallel core maintenance algorithm, so-called Parallel-Order, for both edge insertion and removal.

The main steps for parallel edges in parallel are shown in Algorithm 4. Given an undirected graph G, the core number and k-order can be initialized by the BZ algorithm [1] in linear time. A batch ΔE of edges will insert into G. We split these edges ΔE into $\mathcal{P}$ parts, $\Delta E_1 \dots \Delta E_{\mathcal{P}}$, where $\mathcal{P}$ is the total number of workers (line 1). Each worker p inserts multiple inserted edges of ΔE_p in parallel with other workers (line 2). One by one, a worker p deals with a single edge in $\texttt{InsertEdge}_p$ (line 4). The key issue is how to implement $\texttt{InsertEdge}_p$ executed by a worker p in parallel with other workers.

Removing edges in parallel is analogous to Algorithm 4, and the key issue is $\texttt{RemoveEdge}_p$. Note that insertion and removal cannot run in parallel, which greatly simplifies the synchronization of the algorithms.

One benefit of our method is that, unlike the existing parallel core maintenance methods [13, 14, 29], prepossessing of ΔE_p is not required so that edges can be inserted or removed on-the-fly.

Algorithm 4: Parallel-InsertEdges($G, \mathbb{O}, \Delta E$)

```
1 partition ΔE into ΔE_1, ..., ΔE_P
2 DoInsert_1(ΔE_1) ‖ ··· ‖ DoInsert_P(ΔE_P)
3 procedure DoInsert_p(ΔE_p)
4     for (u, v) ∈ ΔE_p do InsertEdge_p(G, 𝕆, (u, v))
```

4.1 Parallel Edge Insertion

Algorithm. The detailed steps of $\texttt{InsertEdge}_p$ are shown in Algorithm 6, which is analogous to Algorithm 1. We introduce several new data structures. First, the min-priority queue Q_p, the queue R_p, the candidate set V_p^*, and the searching set V_p^+ are all private to each worker p, so cannot be accessed by other workers and synchronization is not necessary (lines 3, 7). Second, for each vertex $u \in V$, we introduce a status $u.s$, initialized as 0, and atomically incremented by 1 before and after the k-order operation (lines 16 and 30). In other words, when $u.s$ is an odd number, the k-order of u is being maintained. By using such a status of each vertex, we obtain $v \in u.post$ ($u \preceq v$) or $v \in u.pre$ ($v \preceq u$) by the parallel $\texttt{Order}(u, v)$ operation. As shown in Algorithm 5, when comparing the order of u and v, we ensure that u and v are not updating their k-order.

Algorithm 5: Parallel-Order($\mathbb{O}, u, v$)

```
1 do
2     do s ← u.s; s' ← v.s while s mod 2 = 0 ∨ s' mod 2 = 0
3     r ← u ⪯ v
4 while s = u.s ∧ s' = v.s
5 return r
```

Given an inserted edge $u \mapsto v$ where $u \preceq v$, we lock both u and v together when both are not locked (line 1). We redo the lock of u and v if they are updated by other workers as $v \preceq u$ (line 2). After locking, K is initialized to the smaller core number of u and v. After inserting the edge $u \mapsto v$ into the graph G (line 4), v can be unlocked (line 5). If $u.d_{out}^+ \le K$, we unlock u and terminate (line 6); otherwise, we set w as u for propagation (line 7). In the do-while-loop (lines 8 - 13), initially, w equals u, which was already locked in line 1 (line 7). We calculate $w.d_{in}^*$ by counting the number of $w.pre$ located in V_p^* (line 9). If $w.d_{in}^* + w.d_{out}^+ > K$, vertex w does $\texttt{Forward}_p$ (line 10). If $w.d_{in}^* + w.d_{out}^+ \le K \wedge w.d_{in}^* > 0$, vertex w does $\texttt{Backward}_p$ (line 11). If $w.d_{in}^* + w.d_{out}^+ \le K \wedge w.d_{in}^* = 0$, we skip w and unlock w since w cannot be in V^+ (line 11). Successively, we dequeue a vertex w from Q_p with $w.core = K$ and lock w at the same time (line 12). The do-while-loop terminates when no more vertices can be dequeued from Q_p (line 13). All vertices $w \in V_p^*$ have their core numbers increased by 1 and their $w.d_{in}^*$ is reset to 0 (line 15); also, all w are removed from $\mathbb{O}_K$ and inserted at the head of $\mathbb{O}_{K+1}$ to maintain the k-order by using the parallel OM data structure, where all $w.s$

are atomically increased by 1 before and after this process (line 16). Before termination, we unlock all locked vertices w (line 17).

The $\texttt{Forward}(u)$ and $\texttt{Backward}(w)$ procedures are almost the same as their sequential version since all vertices in V^+ are locked. There are only several differences. In $\texttt{Forward}_p(u)$, for each v in $u.post$ whose core numbers equal to K, we add v into the priority queue Q_p (line 21); but $v.d^*_{in}$ is not maintained by adding 1 since it will be calculated in line 9 when it is used. In the $\texttt{Backward}_p(w)$ procedure, w is removed from $\mathbb{O}_K$ and appended after pre to maintain the k-order by using the parallel OM data structure, where $w.s$ are atomically increased by 1 before and after this process (line 30).

Example 4.1. Continuing with Figure 1, we show an example of maintaining the core numbers of vertices in parallel after inserting three edges. Figure 1(b) shows three edges, e_1, e_2 and e_3, being inserted in parallel by three workers, p_1, p_2, and p_3, respectively. (1) For e_1, the worker p_1 will first locks v and u_2 for inserting the edge. But if u_2 is already locked by p_2, worker p_1 has to wait for p_2 to finish and unlock u_2. (2) For e_2, worker p_2 first locks u_2 and u_3 for inserting the edge, after which u_3 is unlocked. Then, u_3, u_4, and u_5 are added to its priority queue Q_2 for propagation. That is, u_3 is locked and dequeued from Q_2 with $u_3.d^*_{in} = 1$ (assuming that p_2 locks u_3 before P_3 lock u_3). After propagation, we get that V^* is empty. Subsequently, u_4 and u_5 are locked and dequeued from Q_2, which are unlocked and skipped since their $d^*_{in} = 0$. The k-order $\mathbb{O}_2$ is updated to u_1, u_3, u_2, u_4, and u_5. (3) For e_3, the worker p_3 will first lock u_1 and u_4 for inserting the edge, after which u_4 is unlocked. Then, u_3, u_4 and u_5 are added to Q_3 for propagation. That is, u_3 is locked and dequeued from Q_2 (assuming that p_3 waits for u_3 to be unlocked by p_2) with $u_3.d^*_{in} = 1$, by which u_3 is added to V^* and u_2 is added to Q_3 for propagation. Subsequently, u_3, u_2, u_4, and u_5 are locked and dequeued from Q_3 for propagation, which are all added to V^* (assuming that p_3 waits for u_2 to be unlocked by p_2).

We can see three vertices, u_3, u_4 and u_5, can be added in Q_2 and Q_3 at the same time. That is, when p_3 removes u_3 from Q_2, it is possible that u_3 has already been accessed by p_2. In this case, we have to update Q_3 before dequeuing if we find that u_3 is accessed by p_2, in case the k-order of u_3 in Q_3 is changed by p_2.

Implementation. The min-priority queue Q is used for traversing the affected vertices in k-order $\mathbb{O}$. Here, $\mathbb{O}$ is implemented by the parallel OM data structure [11], in which all vertex are assigned labels to indicate the order. Queue Q is implemented with min-heap [5] by comparing the labels maintained by the parallel OM data structure, which supports enqueue and dequeue in $O(\log |Q|)$ time. For a worker p, all vertices $v \in Q_p$ can be locked and reordered by other workers. To correctly dequeue a vertex that has a minimum order in Q_p, we devise specific enqueue and dequeue operations:

- When enqueuing w into Q_p, we recorded $w.s$. When dequeuing v from Q_p, we first lock v and check if $v.s$ has changed or not. If that is the case, v is reordered and the label of v is changed by other workers, so we have to make the heap of Q_p again and redo the dequeue operation.
- When dequeuing v from Q_p, if $\mathbb{O}_k$ does not trigger a relabel operation (including rebalance and split), the locked v will always have a min-label (smallest order in $\mathbb{O}_k$), since all the other vertices must have their order increased when accessed by other workers (lines 16 and 30 in Algorithm 1).

Algorithm 6: InsertEdge$_p(\vec{G}, \mathbb{O}, u \mapsto v)$

```
1  Lock u and v together when both are not locked
2  if v ≤ u then Unlock u and v; goto line 1
3  V*_p, V+_p, K, ← ∅, ∅, min(u.core, v.core)
4  insert u ↦ v into G⃗ with u.d+_out ← u.d+_out + 1
5  Unlock(v)
6  if u.d+_out ≤ K then Unlock(u); return
7  Q_p, w ← a min-priority queue by 𝕆, u
8  do
9      w.d*_in ← |{w' ∈ w.pre : w' ∈ V*_p}|          // calculate d*_in
10     if w.d*_in + w.d+_out > K then Forward_p(w)
11     else if w.d*_in > 0 then Backward_p(w) else Unlock(w)
12     w ← Q_p.dequeue() with w.core = K and Lock(w)
13 while w ≠ ∅
14 for w ∈ V*_p do
15     w.core ← K + 1; w.d*_in ← 0
       // atomically add w.s
16     ⟨w.s++⟩; Delete(𝕆_K, w); Insert(𝕆_{K+1}, head, w); ⟨w.s++⟩
17 Unlock all locked vertices

18 procedure Forward_p(u)
19     V*_p ← V*_p ∪ {u}; V+_p ← V+_p ∪ {u}          // u is locked
20     for v ∈ u.post : v.core = K do
21         if v ∉ Q_p then Q_p.enqueue(v)

22 procedure Backward_p(w)
23     V+_p ← V+_p ∪ {w}; pre ← w                    // w is locked
24     R_p ← an empty queue; DoPre_p(w, R_p)
25     w.d+_out ← w.d+_out + w.d*_in; w.d*_in ← 0
26     while R_p ≠ ∅ do
27         u ← R_p.dequeue()
28         V*_p ← V*_p \ {u}
29         DoPre_p(u, R_p); DoPost_p(u, R_p)
           // atomically add w.s
30         ⟨w.s++⟩; Delete (𝕆_K, u); Insert(𝕆_K, pre, u); ⟨w.s++⟩
31         pre ← u; u.d+_out ← u.d+_out + u.d*_in; u.d*_in ← 0

32 procedure DoPre_p(u, R_p)
33     for v ∈ u.pre : v ∈ V*_p do
34         v.d+_out ← v.d+_out − 1
35         if v.d*_in + v.d+_out ≤ K ∧ v ∉ R_p then R_p.enqueue(v)

36 procedure DoPost_p(u, R_p)
37     for v ∈ u.post do
38         if v ∈ V*_p ∧ v.d*_in > 0 then
39             v.d*_in ← v.d*_in − 1
40             if v.d*_in + v.d+_out ≤ K ∧ v ∉ R_p then R_p.enqueue(v)
```

- Within the enqueue and dequeue operations, if $\mathbb{O}_k$ triggers a relabel operation, the label of $v \in \mathbb{O}_k$ may be decreased. In this case, we have to make the heap of $\mathbb{O}_k$ again and redo such enqueue and dequeue operations.

Typically, the size of Q_p is small and the relabel operations of $\mathbb{O}_k$ are triggered with a low probability. Thus, our enqueue and dequeue operations remain efficient.

Correctness and Complexities. Please refer to the appendix.

4.2 Parallel Edge Removal

Algorithm. The detailed steps of $\texttt{RemoveEdge}_p$ are shown in Algorithm 7. We introduce several new data structures. First, the queue R_p is privately used by worker p and cannot be accessed by other workers without synchronization (line 2). Second, each worker p adopts a set A_p to record all the visited vertices $w' \in w.adj$ to avoid repeatedly revisiting $w' \in w.adj$ again. Third, each vertex $v \in V$ has a status $v.t$ with four possible values:

- $v.t = 2$ means v is ready to be propagated (line 22).
- $v.t = 1$ means v is been propagated by the inner for-loop (lines 11 - 14).
- $v.t = 3$ means v has to be propagated again by the inner for-loop (lines 11 - 14), as some vertices $v.adj$ have core numbers decreased by other workers.
- $v.t = 0$ means v is just initialized or already propagated (line 33).

Given a removed edge (u, v), we lock both u and v together when both are not locked (line 1). After locking, K is initialized as the smaller core number of u and v (line 2). We execute the procedure $\texttt{CheckMCD}_p$ for u or v to make $u.mcd$ and $v.mcd$ non-empty (line 3). We remove the edge (u, v) safely from the graph G (line 4). For u or v, if their core number is greater or equal to K, we execute the procedure $\texttt{DoMCD}_p$ (lines 5 and 6), by which u and v may be added in R_p for propagation. If u or v is not in R_p, we immediately unlock u or v (line 7). The while-loop (lines 8 - 16) propagates all vertices in R_p. A vertex w is removed from R_p and an empty set A_p is initialized (line 9). In the inner for-loop (lines 11 - 14), the adjacent vertices $w' \in w.adj$ are condition-locked with $w'.core = K$ (lines 11 and 12), as $w'.core$ can be decreased from K to $K-1$ by other workers. For each locked $w' \in w.adj$, we first execute the $\texttt{CheckMCD}_p$ procedure in case $w'.mcd$ is empty and then execute the $\texttt{DoMCD}_p$ procedure (line 13). The visited w' are added into A_p to avoid visiting them repeatedly (line 14). We atomically decrease $w.t$ by 1 before and after such an inner for-loop since other workers can access $w.t$ in line 32 (lines 10 and 15). After that, if $w.t > 0$, we have to propagate w again as other vertices in $w.adj$ have core numbers decreased from $K+1$ to K by other workers (line 16). The while-loop will not terminate until R_p becomes empty (line 8). Finally, all vertices in V^* are appended to $\mathbb{O}_{K-1}$ to maintain the k-order (line 17). We must not forget to unlock all locked vertices before termination (line 18).

In procedure $\texttt{DoMCD}_p(u)$, vertex u has already been locked by worker p (line 19). We decrease $u.mcd$ by 1 as $u.mcd$ cannot be empty (line 20). If it still has $u.mcd \geq u.core$, we finally unlock u and terminate (line 21 and 25). Otherwise, we first decrease $u.core$ by 1 and set $u.t$ as 2 together, which has to be an atomic operation since $v.t$ indicates v's status for other workers (line 22). Then, we add u to R_p for propagation (line 23); also, we set $u.mcd$ to empty since the value is out of date, which can be calculated later if needed (line 24).

In the procedure $\texttt{CheckMCD}(u)$, we recalculate $u.mcd$ if it is empty (line 27). We initially set temporarily mcd as 0 (line 28), and then we count $u.mcd$ (lines 29 - 33). Here, $u.mcd$ is the number of $v \in u.adj$ for two cases: 1) $v.core \geq u.core$, or 2) $v.core = u.core - 1$ and $v.t > 0$ (line 29); if either one is satisfied,we add the temporal mcd by 1 (line 30). When $v.core = K-1$, it is possible that $v.t$ is been updated by other workers. If $v.t$ equals 1, we know that v is been propagating. In this case, we have to set $v.s$ from 1 to 3 by the atomic primitive CAS, which leads to v redo the propagation in line 16 by other workers (line 32). Here, we skip executing CAS when $v = w$ (line 32) to avoid many useless redo processes in line 13. If $v.t$ is reduced to 0, the propagation of v is finished so that v cannot be counted as $u.mcd$ and the temporary mcd is off by 1 (line 33). Finally, we set $u.mcd$ as the temporary mcd and terminate (line 34). The big advantage is that we calculate $u.mcd$ without locking all neighbors $u.adj$ of v.

Algorithm 7: $\texttt{RemoveEdge}_p(G, \mathbb{O}, (u, v))$

```
1  Lock u and v together when both are not locked
2  K, R_p, V_p* ← Min(u.core, v.core), an empty queue, ∅
3  CheckMCD_p(u, ∅); CheckMCD_p(v, ∅)
4  remove (u, v) from G
5  if v.core ≥ K then DoMCD_p(u)
6  if u.core ≥ K then DoMCD_p(v)
7  Unlock u if u ∉ R_p; Unlock v if v ∉ R_p
8  while R_p ≠ ∅ do
9      w, A_p ← R_p.dequeue(), ∅
10     ⟨w.t ← w.t − 1⟩                                  // atomically sub
11     for w' ∈ w.adj : w' ∉ A_p ∧ w'.core = K do
12         if Lock(w') with w'.core = K then
13             CheckMCD_p(w', w); DoMCD_p(w')
14             A_p ← A_p ∪ {w'}
15     ⟨w.t ← w.t − 1⟩                                  // atomically sub
16     if w.t > 0 then goto line 10
17 Append all u ∈ V_p* at the tail of 𝕆_{K−1} in k-order
18 Unlock all locked vertices

19 procedure DoMCD_p(u)
20     u.mcd ← u.mcd − 1                                // u is locked
21     if u.mcd < K then
22         ⟨u.core ← K − 1; u.t = 2⟩                    // atomic operation
23         R_p.enqueue(u); u.mcd ← ∅
24         V_p* ← V_p* ∪ {u}; Delete(𝕆, u)
25     else Unlock(u)

26 procedure CheckMCD_p(u, w)
27     if u.mcd ≠ ∅ then return
28     mcd ← 0
29     for v ∈ u.adj : v.core ≥ K ∨ (v.core = K − 1 ∧ v.t > 0) do
30         mcd ← mcd + 1
31         if v.core = K − 1 then
32             if v ≠ w ∧ v.t = 1 then CAS(v.t, 1, 3)
33             if v.t = 0 then mcd ← mcd − 1
34     u.mcd ← mcd
```

Example 4.2. Continuing with Figure 2, we show an example of maintaining the core numbers of vertices in parallel when removing three edges. Figure 2(b) shows three edges, e_1, e_2 and e_3, being removed in parallel by three workers, p_1, p_2, and p_3, respectively. (1) For e_1, worker p_1 will lock v and u_2 together for removing the edge. But u_2 is already locked by p_2, so p_1 has to wait for p_2 to unlock u_2. Then, u_2 is unlocked without changing $u_2.mcd$, and the core number of v is off by 1 added to R_1 for propagation. Since only one $u_3 \in v.adj$ has a core number greater than v, the propagation of v terminates. Finally, v is unlocked. (2) For e_2, the worker p_2

first locks u_2 and u_3 together for removing the edge. Then, both $u_2.core$ and $u_3.core$ are off by 1, and u_2 and u_3 are added to R_2 for propagation. For propagating u_2, we traverse all $u_2.adj$; the vertex u_4 is locked by the worker p_3. At the same time, $u_4.core$ is decreased from 2 to 1 and p_1 will skip locking u_4 since the condition is not satisfied for the conditional lock. Vertex u_5 is locked by p_2 and has $u_5.mcd$ off by 1. Similarly, for propagating u_3, we traverse all $u_3.adj$ by skipping u_1 and decreasing $u_5.mcd$. Now, we have $u_5.mcd = 2 < u_5.core = 3$, so $u_5.core$ is off by 1. Finally, we unlock u_2, u_3, and u_5; all their core numbers are 2 now. (3) For e_3, worker p_3 will first lock u_1 and u_4 together for removing the edge. Then both $u_1.core$ and $u_4.core$ are off by 1. Vertices u_1 and u_4 are added to R_3 for propagation. The propagation will stop since the neighbors of u_1 and u_4 (u_3, u_2, and u_5) are locked by p_2 and have decreased core numbers. Finally, we unlock u_1 and u_4; all their core numbers are 2 now. We can see p_2 and p_3 execute without blocking each other, and only vertices in V^* are locked.

The above example assumes that the mcd of all vertices is initially generated. Suppose $u_3.mcd = \emptyset$ before removing e_2, we have to calculate $u_3.mcd$ by `CheckMCD`. At this time, u_2 and u_5 are counted as $u_3.mcd$ since they are not locked by p_3, but u_1 is locked by p_3 for propagation. The key issue is whether u_1 is counted as $u_3.mcd$ or not. There are two cases. (1) If $u_1.core = 3$, we increment $u_3.mcd$ by 1. (2) If $u_1.core$ is decreased to 2 and u_1 is propagating, we also increment $u_3.mcd$ by 1. Since it is possible that u_1 has already propagated u_3, we force u_1 to redo the propagation by setting $u_1.t$ from 1 to 3 atomically.

Correctness and Complexities. Please refer to the appendix.

5 Experiments

In this section, we experimentally compare the following core maintenance approaches:

- The *Join Edge Set* based parallel edge insertion algorithm (`JEI` for short) and removal algorithm (`JER` for short) [13]
- The *Matching Edge Set* based parallel edge insertion (`MI` for short) and removal algorithm (`MR` for short) [14]
- Our parallel edge insertion algorithm (`OurI` for short) and removal algorithm (`OurR` for short)
- As baselines, the sequential Simplified-Order edge insertion algorithm (`OI` for short) and removal algorithm (`OR` for short) [12]
- As baselines, the sequential Traveral edge insertion algorithm (`TI` for short) and removal algorithm (`TR` for short) [26]

The source code is available on GitHub[1].

Experiment Setup. The experiments are performed on a server with an AMD CPU (64 cores, 128 hyperthreads, 256 MB of last-level shared cache) and 256 GB of main memory. Each core corresponds to a worker. The server 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 perform every experiment at least 50 times and calculate their means with 95% confidence intervals.

Tested Graphs. We evaluate the performance of different methods over a variety of real-world and synthetic graphs shown in Table 1. For simplicity, directed graphs are converted to undirected ones; all of the self-loops and repeated edges are removed. That is, a vertex cannot connect to itself, and each pair of vertices can connect with at most one edge. The *livej*, *patent*, *wiki-talk*, and *roadNet-CA* graphs are obtained from SNAP[3]. The *dbpedia*, *baidu*, *pokec* and *wiki-talk-en* *wiki-links-en* graphs are collected from the KONECT[4] project. The *ER*, *BA*, and *RMAT* graphs are synthetic graphs generated by the SNAP[5] system using Erdös-Rényi, Barabasi-Albert, and the R-MAT graph models, respectively; the average degree is fixed to 8 by choosing 1,000,000 vertices and 8,000,000 edges. All the above twelve graphs are static graphs. We randomly sample 100,000 edges for insertion and removal.

We also select four real temporal graphs, *DBLP*, *Flickr*, *StackOverflow*, and *wiki-edits-sh* from KONECT; each edge has a timestamp recording the time of this edge inserted into the graph. We select 100,000 edges within a continuous time range for insertion and removal.

Graph	$n = \|V\|$	$m = \|E\|$	AvgDeg	Max k
livej	4,847,571	68,993,773	14.23	372
patent	6,009,555	16,518,948	2.75	64
wikitalk	2,394,385	5,021,410	**2.10**	131
roadNet-CA	1,971,281	5,533,214	2.81	**3**
dbpedia	3,966,925	13,820,853	3.48	20
baidu	2,141,301	17,794,839	8.31	78
pokec	1,632,804	30,622,564	18.75	47
wiki-talk-en	2,987,536	24,981,163	8.36	210
wiki-links-en	5,710,993	130,160,392	22.79	**821**
ER	1,000,000	8,000,000	8.00	11
BA	1,000,000	8,000,000	8.00	8
RMAT	1,000,000	8,000,000	8.00	237
DBLP	1,824,701	29,487,744	16.17	286
Flickr	2,302,926	33,140,017	14.41	600
StackOverflow	2,601,977	63,497,050	**24.41**	198
wiki-edits-sh	4,589,850	40,578,944	8.84	47

Table 1: Tested real and synthetic graphs.

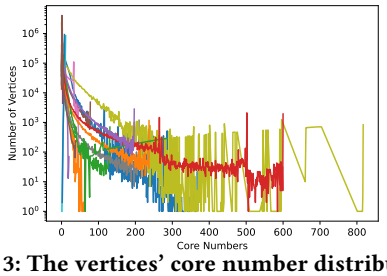

Figure 3: The vertices' core number distributions.

In Table 1, we can see all graphs have millions of edges. Their average degrees range from 2.1 to 24.4, and their maximal core numbers range from 3 to 821. In Figure 3, we can see that the core numbers of vertices are not uniformly distributed in all tested graphs, where the x-axis is core numbers and the y-axis is the number of vertices. That is, a great portion of vertices have small core numbers, and few have large core numbers. For example, *wikitalk* has 1.7 million vertices with a core number of 1; *roadNet-CA* has four core numbers from 0 to 3; *BA* only has a single core number

[1] https://github.com/Itisben/Parallel-CoreMaint.git
[2] https://www.openmp.org/
[3] http://snap.stanford.edu/data/index.html
[4] http://konect.cc/networks/
[5] http://snap.stanford.edu/snappy/doc/reference/generators.html

of 8. For JEI, JER, MI and MR, such core number distribution is an important property since the vertices with the same core number can only be processed by a single worker at the same time, while OurI and OurR do not have this limitation.

5.1 Running Time Evaluation

In this experiment, we exponentially increase the number of workers from 1 to 64 to evaluate the real running time over graphs in Table 1. For the twelve static graphs, we randomly sample 100,000 edges. For the four temporal graphs, we select the latest period of 100,000 edges. These edges are first removed and then inserted. The accumulated running times are measured.

The plots in Figure 4 depict the performance of four compared algorithms, where the running times above 3600 seconds are not depicted. Comparing three parallel methods, the first look reveals that OurI and OurR always have the best performance and MI and MR always have the worst performance, respectively. Compared with the two baseline methods, we find that OI and OR are much more efficient than TI and TR, respectively. Specifically, we make several observations:

- By using one worker, Our and OurR have the same running time as the baselines of OI and OR, respectively. This is because OurI and OurR are based on OI and OR and have the same work complexities, respectively.
- By using one worker, JEI and JER are always faster than TI and TR, respectively. This is because although JEI and JER are based on TI and TR, a batch of insertions or removals are processed together and thus repeated computations can be avoided. Also, MI and MR have the same trend.
- By using one worker, all algorithms are reduced to sequential, and OurI performs much faster than JEI. This is because for edge insertion, OurI is based on the OI, while JEI is based on TI. OI is much faster than TI. Also, MI and MR have the same trend.
- By using one worker, OurR does not always perform better than JER. This is because our method uses arrays to store edges, which can save space, while the join-edge-set-based method uses binary search trees to store edges. When deleting an edge (u, v), OurR has to traverse all vertices of $u.adj$ and $v.adj$, while JER only need to traverse $\log |u.adj|$ and $\log |v.adj|$ vertices. That means OurR costs more running time than JER for deleting an edge from the graph.
- By using multiple workers, OurI and OurR can always achieve better speedups compared with other parallel methods, but JEI and JER have no speedups over some graphs. This is because JEI and JER have limited parallelism, as affected vertices with different core numbers cannot perform in parallel, while OurI and OurR do not have such a limitation. Also, MI and MR have the same trend.
- By using multiple workers, the running time of OurI and OurR may begin to increase when using more than 8 or 16 workers, e.g. *livej*, *patent*, and *dbpedia*. This is because of the contention on shared data structures with multiple workers, and more workers may lead to higher contention. In addition, for JEI and JER, when the core numbers of vertices in graphs are not well distributed, some workers are wasted, which results in extra overheads.

In Table 2, columns 2 to 7 compare the running time speedups between using 1 worker and 16 workers for all tested algorithms. It is clear that OurI and OurR always achieve better speedups up to 5x, compared with other parallel methods. Columns 8 to 11 compare the running time speedups between our method and the other parallel methods by using 1 worker; columns 12 to 15 compare the running time speedups between our method and the other parallel methods by using 16 workers. We first can see that compared with MI and MR, OurI and OurR always achieve the highest speedups both using 1 worker and 16 workers. Compared with JEI, OurI achieves up to 50x speedups even using 1 worker and achieves up to 289x speedups when using 16 workers. We observe that compared with JER, OurR does not always achieve speedups when using a single worker, but achieves up to 10x speedups when using 16 workers. Especially, over *wiki-edits-sh*, OurI and OurR run slightly slower than JEI and JER when using 1 worker and 16 workers, respectively. The reason is that the special properties of graphs may affect the performance of different algorithms.

5.2 Locked Vertices Evaluation

In Figure 5, for OurI and OurR, we summarize the number of different sizes of V^+ for the last experiment (Section 5.1). The x-axis is the size of V^+ and the y-axis is the number of such size of V^+. We observe that more than 97% of inserted or removed edges have $|V^+|$ between 0 and 10. That means, there is a high probability that less or equal to 10 vertices are locked when inserting or removing one edge. Less locked vertices will lead to high parallelism

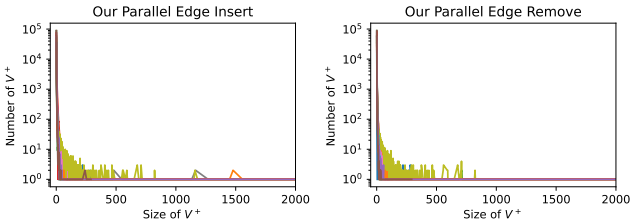

Figure 5: The size of V^+ for OurI and OurR.

6 Conclusions and Future Work

We present new parallel core maintenance algorithms to handle a batch of inserted or removed edges based on the Order algorithm. A set of vertices V^+ are traversed. We use locks for synchronization. Only the vertices in V^+ are locked and all their associated edges are not necessarily locked, which leads to high parallelism.

The proposed parallel methodology can be applied to other graphs, e.g. weighted graphs and probability graphs. It can also be applied to other graph algorithms, e.g. maintaining the k-truss in dynamic graphs. Additionally, the maintenance of the hierarchical k-core involves maintaining the connections among different k-cores in the hierarchy, which can benefit from our result.

References

[1] Vladimir Batagelj and Matjaz Zaversnik. An $O(m)$ algorithm for cores decomposition of networks. *CoRR*, cs.DS/0310049, 2003.

[2] Michael A Bender, Richard Cole, Erik D Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In *European Symposium on Algorithms*, pages 152–164. Springer, 2002.

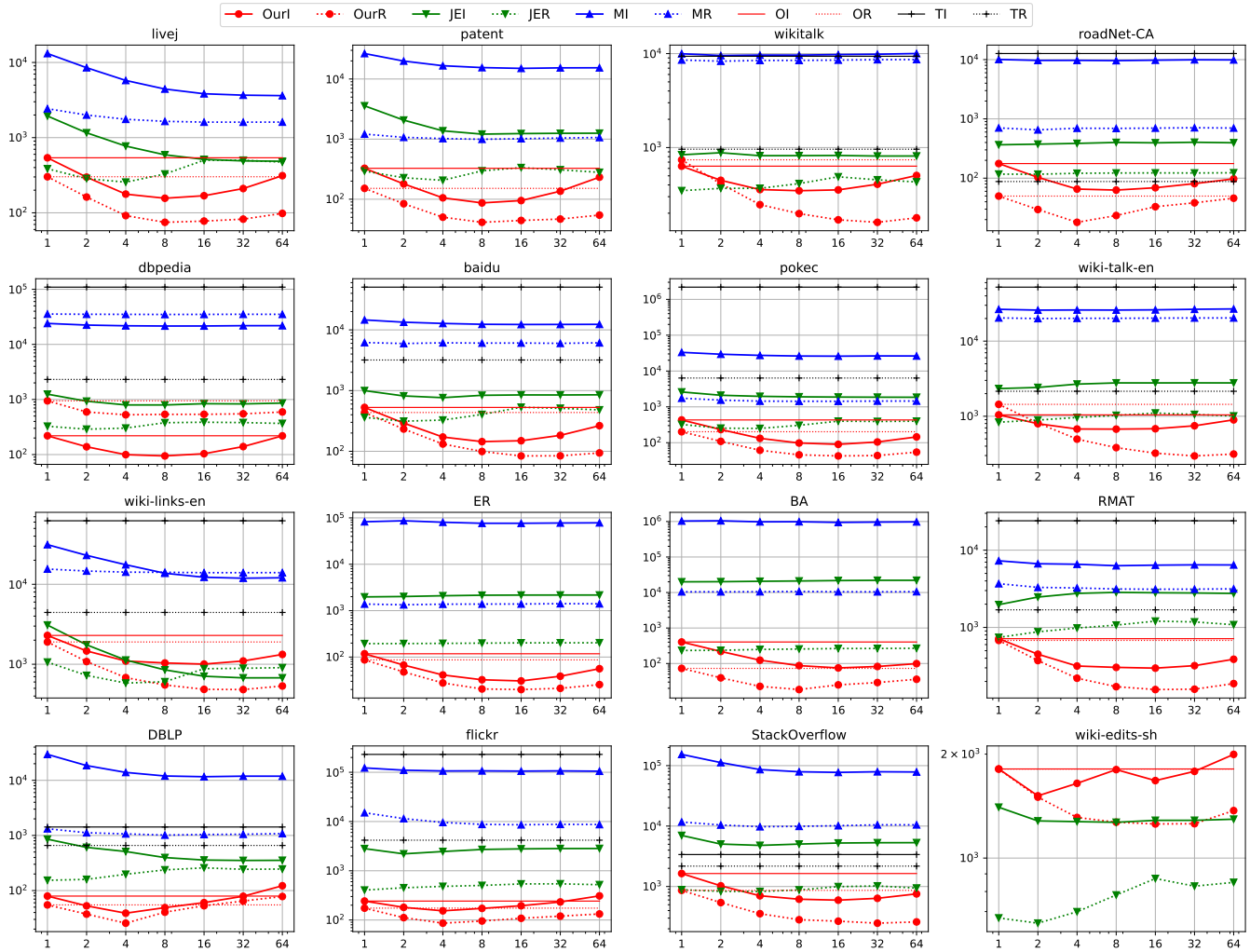

Figure 4: The real running time by varying the number of workers. The x-axis is running time (millisecond) and the y-axis is the number of workers.

Graph	1-worker vs 16-worker						1-worker OurI vs		1-worker OurR vs		16-worker OurI vs		16-worker OurR vs	
	OurI	OurR	JEI	JER	MI	MR	JEI	MI	JER	MR	JEI	MI	JER	MR
livej	3.2	3.9	3.8	0.8	**3.4**	1.5	3.6	24.4	0.7	4.5	3.0	22.7	3.0	**9.5**
patent	3.4	3.4	2.9	0.9	1.8	1.2	11.0	81.2	0.9	**3.7**	13.0	158.3	3.5	10.7
wikitalk	1.8	4.4	1.0	0.7	**1.0**	**1.0**	1.3	15.7	0.5	13.5	2.3	27.6	1.4	24.1
roadNet-CA	2.6	1.5	0.9	1.0	1.0	1.0	2.1	57.1	0.7	4.0	5.7	141.9	1.8	10.1
dbpedia	2.1	1.8	1.5	0.8	1.1	1.0	5.7	109.4	1.5	**162.0**	8.1	208.1	3.7	**337.9**
baidu	3.5	**5.2**	1.2	0.7	1.2	1.0	1.9	27.6	0.7	11.7	5.7	82.1	3.6	40.7
pokec	4.8	4.7	1.4	0.8	1.3	1.2	6.0	76.9	0.7	4.0	20.9	288.8	4.4	15.9
wiki-talk-en	1.5	4.5	0.8	0.8	1.0	1.0	2.2	25.4	0.8	19.5	4.1	38.2	1.6	29.7
wiki-links-en	2.3	3.9	**4.4**	**1.2**	2.6	1.1	1.3	13.7	0.5	6.8	**0.7**	**12.2**	0.9	13.9
ER	3.8	4.4	0.9	1.0	1.1	1.0	16.8	700.3	1.7	11.8	70.9	2500.7	**6.6**	45.5
BA	**5.4**	2.9	0.9	0.9	1.1	1.0	**49.6**	**2555.3**	0.6	25.9	**289.1**	**12552.9**	3.5	139.7
RMAT	2.4	4.3	**0.7**	**0.6**	1.1	1.2	2.8	**10.2**	1.0	5.2	9.5	21.5	4.1	10.5
DBLP	1.3	**1.0**	2.4	0.6	2.5	1.3	10.8	371.7	**1.9**	16.7	5.9	192.4	4.3	17.2
flickr	1.2	1.6	1.0	0.7	1.2	**1.8**	11.6	506.7	1.7	62.3	14.3	542.3	2.8	44.1
StackOverflow	2.8	3.2	1.3	0.9	2.0	1.2	4.3	93.5	0.5	7.1	8.8	130.0	1.7	17.1
wiki-edits-sh	**1.1**	1.4	1.1	0.8	-	-	**0.8**	-	**0.4**	-	0.8	-	**0.5**	-

Table 2: Compare the speedups.

[3] Kate Burleson-Lesser, Flaviano Morone, Maria S Tomassone, and Hernán A Makse. k-core robustness in ecological and financial networks. *Scientific reports*, 10(1):1–14, 2020.

[4] James Cheng, Yiping Ke, Shumo Chu, and M Tamer Özsu. Efficient core decomposition in massive networks. In *2011 IEEE 27th International Conference on Data Engineering*, pages 51–62. IEEE, 2011.

[5] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[6] Naga Shailaja Dasari, Ranjan Desh, and Mohammad Zubair. Park: An efficient algorithm for k-core decomposition on multicore processors. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 9–16. IEEE, 2014.

[7] Paul Dietz and Daniel Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 365–372, 1987.

[8] Rodrigo Dorantes-Gilardi, Diana García-Cortés, Enrique Hernández-Lemus, and Jesús Espinal-Enríquez. k-Core genes underpin structural features of breast cancer. *Scientific Reports*, 11(1):1–17, 2021.

[9] Kasimir Gabert, Ali Pinar, and Ümit V Çatalyürek. Shared-memory scalable k-core maintenance on dynamic graphs and hypergraphs. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 998–1007. IEEE, 2021.

[10] Pengqun Gao, Jing Huang, and Yejun Xu. A k-core decomposition-based opinion leaders identifying method and clustering-based consensus model for large-scale group decision making. *Computers & Industrial Engineering*, 150:106842, 2020.

[11] Bin Guo and Emil Sekerinski. New parallel order maintenance data structure. *arXiv preprint arXiv:2208.07800*, 2022.

[12] Bin Guo and Emil Sekerinski. Simplified algorithms for order-based core maintenance. *arXiv preprint arXiv:2201.07103*, 2022.

[13] Qiang-Sheng Hua, Yuliang Shi, Dongxiao Yu, Hai Jin, Jiguo Yu, Zhipen Cai, Xiuzhen Cheng, and Hanhua Chen. Faster parallel core maintenance algorithms in dynamic graphs. *IEEE Transactions on Parallel and Distributed Systems*, 31(6):1287–1300, 2019.

[14] Hai Jin, Na Wang, Dongxiao Yu, Qiang Sheng Hua, Xuanhua Shi, and Xia Xie. Core Maintenance in Dynamic Graphs: A Parallel Approach Based on Matching. *IEEE Transactions on Parallel and Distributed Systems*, 29(11):2416–2428, nov 2018.

[15] Humayun Kabir and Kamesh Madduri. Parallel k-core decomposition on multicore platforms. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1482–1491. IEEE, 2017.

[16] Wissam Khaouid, Marina Barsky, Venkatesh Srinivasan, and Alex Thomo. k-core decomposition of large networks on a single pc. *Proceedings of the VLDB Endowment*, 9(1):13–23, 2015.

[17] Yi-Xiu Kong, Gui-Yuan Shi, Rui-Jie Wu, and Yi-Cheng Zhang. k-core: Theories and applications. *Physics Reports*, 832:1–32, 2019.

[18] Rong-Hua Li, Jeffrey Xu Yu, and Rui Mao. Efficient core maintenance in large dynamic graphs. *IEEE Transactions on Knowledge and Data Engineering*, 26(10):2453–2465, 2013.

[19] Zhe Lin, Fan Zhang, Xuemin Lin, Wenjie Zhang, and Zhihong Tian. Hierarchical core maintenance on large dynamic graphs. *Proceedings of the VLDB Endowment*, 14(5):757–770, 2021.

[20] Bin Liu and Feiteng Zhang. Incremental algorithms of the core maintenance problem on edge-weighted graphs. *IEEE Access*, PP:1–1, 04 2020.

[21] Fragkiskos D Malliaros, Christos Giatsidis, Apostolos N Papadopoulos, and Michalis Vazirgiannis. The core decomposition of networks: Theory, algorithms and applications. *The VLDB Journal*, 29(1):61–92, 2020.

[22] Daniele Miorandi and Francesco De Pellegrini. K-shell decomposition for dynamic complex networks. In *8th International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks*, pages 488–496. IEEE, 2010.

[23] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. Distributed k-core decomposition. *IEEE Transactions on parallel and distributed systems*, 24(2):288–300, 2012.

[24] Sen Pei, Lev Muchnik, José S Andrade Jr, Zhiming Zheng, and Hernán A Makse. Searching for superspreaders of information in real-world social media. *Scientific reports*, 4:5547, 2014.

[25] Ahmet Erdem Saríyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V Çatalyürek. Streaming algorithms for k-core decomposition. *Proceedings of the VLDB Endowment*, 6(6):433–444, 2013.

[26] Ahmet Erdem Sarıyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V Çatalyürek. Incremental k-core decomposition: algorithms and evaluation. *The VLDB Journal*, 25(3):425–447, 2016.

[27] Bintao Sun, T-H Hubert Chan, and Mauro Sozio. Fully dynamic approximate k-core decomposition in hypergraphs. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 14(4):1–21, 2020.

[28] Robert Utterback, Kunal Agrawal, Jeremy T Fineman, and I-Ting Angelina 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*, pages 83–94, 2016.

[29] Na Wang, Dongxiao Yu, Hai Jin, Chen Qian, Xia Xie, and Qiang-Sheng Hua. Parallel algorithm for core maintenance in dynamic graphs. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 2366–2371. IEEE, 2017.

[30] Ping Wang, Xingdong Deng, Yang Liu, Liang Guo, Jun Zhu, Lin Fu, Yakun Xie, Weilian Li, and Jianbo Lai. A knowledge discovery method for landslide monitoring based on k-core decomposition and the louvain algorithm. *ISPRS International Journal of Geo-Information*, 11(4):217, 2022.

[31] Dong Wen, Lu Qin, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. I/o efficient core graph decomposition at web scale. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 133–144. IEEE, 2016.

[32] Tongfeng Weng, Xu Zhou, Kenli Li, Peng Peng, and Keqin Li. Efficient distributed approaches to core maintenance on large dynamic graphs. *IEEE Transactions on Parallel and Distributed Systems*, 33(1):129–143, 2021.

[33] Huanhuan Wu, James Cheng, Yi Lu, Yiping Ke, Yuzhen Huang, Da Yan, and Hejun Wu. Core decomposition in large temporal graphs. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 649–658. IEEE, 2015.

[34] Michael Yu, Dong Wen, Lu Qin, Ying Zhang, Wenjie Zhang, and Xuemin Lin. On querying historical k-cores. *Proceedings of the VLDB Endowment*, 14(11):2033–2045, 2021.
[35] Feiteng Zhang, Bin Liu, and Qizhi Fang. Core decomposition, maintenance and applications. In *Complexity and Approximation*, pages 205–218. Springer, 2020.
[36] Yikai Zhang, Jeffrey Xu Yu, Ying Zhang, and Lu Qin. A fast order-based approach for core maintenance. In *Proceedings - International Conference on Data Engineering*, pages 337–348, 2017.
[37] Wei Zhou, Hong Huang, Qiang-Sheng Hua, Dongxiao Yu, Hai Jin, and Xiaoming Fu. Core decomposition and maintenance in weighted graph. *World Wide Web*, 24(2):541–561, 2021.

A Parallel Edge Insertion

A.1 Correctness

We only argue the correctness of Algorithm 1 related to the concurrent part. There are no deadlocks since both u and v are locked together for an inserted edge $u \mapsto v$ (line 1) and the propagated vertices are locked in k-order (line 12).

For each worker p, the accessed vertices are synchronized by locking. The key issue is to ensure that a vertex w is locked and then dequeued from Q_p in k-order in the do-while-loop (lines 8 - 12). The invariant is that w has a minimal k-order in Q_p:

$$\forall v \in Q_p : w \notin Q_p \wedge w \preceq v$$

Initially, the invariant is preserved as $w = u$ and $Q_p = \emptyset$. When dequeuing w from Q, the worker p will first lock w that has the minimum k-order and then remove w. In this case, other vertices $v \in Q$ can be accessed by other workers q. For this, there are two cases. 1) Other vertices v may have increased core numbers, which will be removed from Q and skipped. 2) Other vertices v may have $v.d^*_{in} + v.d^+_{out} \leq K$ and cannot be added to V^*_q, which may cause other vertices v' to be removed from V^*_q by the `Backward` procedure; also, all v' are moved after v in k-order, and all v' cannot possibly be moved before v. In a word, all vertices in Q_p cannot have a smaller k-order than w when w is locked.

The worker p traverses $u.post$ in the for-loop (lines 20 - 21, 37 - 40), where u is locked by p; but, all $u.post$ are not locked by p and may be locked by other workers for updating, so do $u.pre$ in the for-loop (lines 33 - 35). The invariant is that all $u.post$ have k-order greater than u and all $u.pre$ have k-order less than u:

$$(v \in u.post \implies u \preceq v) \wedge (v' \in u.pre \implies v' \preceq u)$$

- $v \in u.post \implies u \preceq v$ is preserved as all vertices $u.post$ may have increased core numbers, but v will never be moved before u in k-order by other workers q by the `Backward` procedure, which has been proved before.
- $v' \in u.pre \implies v' \preceq u$ is preserved as u is already locked by worker p so that no other workers can access u and move v' after u in k-order by the `Backward` procedure.

In other words, the sets $u.post$ and $u.pre$ will not change until u is unlocked, even when other workers access the vertices in $u.post$ and $u.pre$.

A.2 Complexities

Time Complexity. When m' edges are inserted into the graphs, the total work is the same as the sequential version in Algorithm 1, which is $O(m'|E^+|\log|E^+|)$, where E^+ is the largest number of adjacent edges for all vertices in V^+ among each inserted edge, defined as $E^+ = \sum_{v \in V^+_p} v.deg$. In the best case, m' edges can be inserted in parallel by $\mathcal{P}$ workers with a depth $O(|E^+|\log|E^+| + m'|V^*|)$ as each worker will not be blocked by other workers; but, all vertices in $|V^*|$ are removed from $\mathbb{O}_K$ and inserted sequentially at the head of O_{K+1}. Therefore, The best-case running time is $O(m'|E^+|\log|E^+|/\mathcal{P} + |E^+|\log|E^+| + m'|V^*|)$. In the worst case, m' edges have to be inserted one by one, which is the same as the sequential execution, since $\mathcal{P}$ workers make a blocking chain. Therefore, the worst-case running time is $O(m'|E^+|\log|E^+|)$.

However, in practice, such a worst-case is unlikely to happen. The reason is that, given a large number of inserted edges, they have a low probability of connecting with the same vertex; also each inserted edge has a small size of V^+ (e.g. 0 or 1) with a high probability.

Space Complexity. For each vertex $v \in V$, it takes $O(3)$ space to store $v.d^*_{in}$, $v.d^+_{out}$, $v.s$, and locks, which makes $O(3n)$ space in total. Each worker p maintains their private V^*_p, V^+_p, which takes $O(2|V^+|\mathcal{P})$ space in total. Similarly, each worker p maintains Q_p and R_p, which take $O(|E^+|\mathcal{P})$ space in total since at most $O(2|E^+|)$ vertices can be added to Q_p and R_p for each inserted edge. The OM data structure is used to maintain the k-order for all vertices in the graph, which takes $O(n)$ space. Therefore, the total space complexity is $O(n + |V^+| + |E^+|\mathcal{P}) = O(n + |E^+|\mathcal{P})$.

B Parallel Edge Removal

B.1 Correctness

Algorithm 7 has no deadlocks. First, both u and v are locked together for a removed edge (u, v) (line 1). Second, for all vertices $w \in R_p$, we have w locked by the worker p and $w.core = K - 1$; also, worker p will lock all $w' \in w.adj$ with $w.core = K$ for propagation (lines 11 and 12). There are four cases:

- if all w' are not locked, there are no deadlocks;
- if w' is locked by another worker q but $w'.core$ is not decreased, there are no deadlocks as w' has no propagation and worker p will wait until w' is unlocked by q;
- if w' is locked by another worker q and w' always has $w'.core > K$, there are no deadlocks as w' is skipped for traversing.
- importantly, if w' is locked by the other worker q and $w'.core$ is decreased from K to $K - 1$, there are no deadlocks as w has propagation stopped on w' for $w'.core = K - 1$ and w' has propagation stopped on w for $w.core = K - 1$. We use "`Lock with`" to conditionally lock w' with $w'.core = K$, which ensure to stop busy-waiting when $w'.core$ decreases from K to $K - 1$.

The key issue of Algorithm 7 is to correctly maintain the *mcd* of all vertices in the graph:

$$\forall v \in V : v.mcd = |\{w \in v.adj : w.core \geq v.core\}| \quad (1)$$

All vertices v in the graph satisfy $v.mcd \geq v.core$; when removing an edge, v with $v.mcd < v.core$ are repeatedly added to V^*_p and have their core numbers decreased by 1 in order to make $v.mcd \geq v.core$. After deleting one edge, the vertices with decreased core numbers

are added into R_p for propagation. The key issue is to argue the correctness of the while-loop (lines 8 to 16) for propagation.

We first define some useful notations. For all vertices $v \in V$, we use $v.lock$, a boolean value, to denote v is locked and $\neg v.lock$ to denote v is unlocked. We use R to denote the union of all propagation queues R_p, denoted as $R = \cup_{p=1}^{\mathcal{P}} R_p$, and a vertex $v \in R$ indicates v can be in one of the R_p for worker p.

The invariant of the outer while-loop (lines 8 - 16) is that all vertices $w \in V$ maintain a status $w.t$ indicating w in R or not; for all vertices $w \in R_p$, which are locked and added into V^*, their core numbers are off by 1 and mcd set as empty (waiting to be recalculated); also, for all vertices $w \in V$, if $w.mcd$ is not empty, $w.mcd$ is the number of neighbors u that have core numbers that are 1) greater or equal $w.core$, or 2) equal to $w.core - 1$ with u in R waiting to be propagated:

$$
\begin{aligned}
&(\forall w \in V : (w.t > 0 \equiv w \in R) \wedge (w.t = 0 \equiv w \notin R)) \\
&\wedge\ (\forall w \in R_p : w.core = K - 1 \wedge w.mcd = \varnothing \wedge w \in V_p^* \\
&\qquad \wedge\ w.lock \wedge w.t > 0) \\
&\wedge\ (\forall u \in V : u.mcd \neq \varnothing \implies u.mcd = |\{v \in u.adj : v.core \geq \\
&\qquad u.core\ \vee\ (v.core = u.core - 1 \wedge v \in R)\}|)
\end{aligned}
$$

The invariant initially holds as vertices u and v may be added to R_p due to deleting an edge (u, v) and u or v is locked if added into R_p. We now argue the while-loop preserve the invariant:

- $\forall w \in V : (w.s > 0 \equiv w \in R) \wedge (w.s = 0 \equiv w \notin R)$ is preserved as $w.t$ is set to 2 and w is added to R at the same time by the atomic operation in line 22; also, $w.s$ is off to 0 when w is removed from R_p for propagation.
- $\forall w \in R_p : w.core = K-1 \wedge w.mcd = \varnothing \wedge w \in V_p^* \wedge w.lock \wedge w.s > 0$ is preserved as when adding w to R_p, $w.core$ is off by 1, $w.mcd$ is set to empty, $w.t$ is set to 2, and w is added to V^*; also, w is locked before added into R.
- $\forall u \in V : u.mcd \neq \varnothing \implies u.mcd = |\{v \in u.adj : v.core \geq u.core \vee (v.core = u.core - 1 \wedge v \in R)\}|$ is preserved as when $u.mcd$ are calculated by the `CheckMCD` procedure, u may have neighbors $v \in u.adj$ whose core numbers are off by 1 and added to R by other workers and the propagation has not yet happened. Note that, the atomic operation in line 22 ensures that u has the core number off by 1 and added to R at the same time.

At the termination of the while-loop, the propagation queue $R = \emptyset$, so that all vertices $w \in V$ have $w.mcd$ correctly maintained.

We now argue the correctness of the inner for-loop (lines 11 - 14), which is important to parallelism. There are two more issues with the inner for-loop. One is that $w'.core$ may be decreased from $K + 1$ to K concurrently by other workers after visiting w' (line 11), which may lead to some w' that have $w'.core$ decreased to K to be skipped. The other is that $v.core$ and $v.s$ may be updated concurrently by other workers (line 29).

We first define useful notations as follow. For the inner for-loop (lines 11 - 14), we denote the set of visited neighbors of w as $w.V$, so that $w.V = \emptyset$ before the for-loop, $w.V \subseteq w.adj$ when executing the for-loop, and $w.V = w.adj$ after the for-loop; also, we denote the set of A_p as $w.A_p$. Note that, we redo the for-loop if $w.t > 0$ by resetting $w.V$ to empty (line 16). We use V^* to denote the union of all V_p^*, denoted as $V^* = \cup_{p=1}^{\mathcal{P}} V_p^*$, and vertex $v \in V^*$ indicates v can be in one of the V_p^* for worker p.

Of course, the outer while-loop (lines 8 - 16) invariant is preserved. The additional invariant of the inner for-loop is that for all vertices $u \in V$, if $u.mcd$ is not empty, $u.mcd$ is the number of neighbors v that have core numbers that are 1) greater or equal to $u.core$, 2) equal to $u.core - 1$ with $u \in R$, or 3) $w.core - 1$ which has u removed from R for propagation but v is not yet propagated by u; also, a status of $v.t = 1$ indicates that v is doing the propagation and $v.t = 0$ indicates that v has finished the propagation:

$$
\begin{aligned}
&\forall w \in V : (w.t = 1 \vee w.t = 3 \equiv w.V \subseteq w.adj) \\
&\wedge\ (\forall u \in V : u.mcd \neq \varnothing \implies u.mcd = |\{v \in u.adj : \\
&\qquad v.core \geq u.core\ \vee\ (v.core = u.core - 1 \wedge v \in R) \vee \\
&\qquad (v.core = u.core - 1 \wedge v \notin R \wedge v \in V^* \wedge v \notin u.V \wedge v \notin u.A_p)\}|)
\end{aligned}
$$

The invariant initially holds as we have $w.t = 1 \wedge w.V = \emptyset \wedge w.A_p = \emptyset$. We now argue that the inner for-loop preserves the invariant:

- $\forall w \in V : (w.t = 1 \vee w.t = 3 \equiv w.V \subseteq w.adj)$ is preserved as $w.t$ is set to 2 when w is added to R_p and $w.s$ is atomically off by 1 before and after the for-loop; also, $w.t$ may be atomically added by 2 by `CAS` when a neighbor w' in $w.adj$ is calculating its mcd.
- $\forall u \in V : u.mcd \neq \varnothing \implies u.mcd = |\{v \in u.adj : v.core \geq u.core \vee (v.core = u.core - 1 \wedge v \in R) \vee (v.core = u.core - 1 \wedge v \notin R \wedge v \in V^* \wedge v \notin u.V \wedge v \notin A_p)$ is preserved as when $u.mcd$ is calculated by the `CheckMCD` procedure, u may have neighbors $v \in u.adj$ whose core numbers are off by 1 and added to R for further propagation; also, it is possible that v is added to V^* and already been removed from R before the inner for-loop (line 9) for propagation, which has three cases:
  - if v not yet traversed u such that $v.core = u.core - 1$ for propagation as $u \notin v.A_p$, u should count v as $u.mcd$.
  - if v has already traversed u such that $v.core = u.core - 1$ for propagation as $u \in v.A_p$, u should not count v as $u.mcd$.
  - if v has already traversed u such that $v.core \neq u.core - 1$ for propagation, but after that $u.core$ has been updated to $v.core = u.core - 1$, u should count v as $u.mcd$.

  The third case requires the repeated traversing of u. We use $v.t = 1$ to let u know that v is executing the inner for-loop (lines 11 - 14) for propagating all vertices in $v.adj$. When $v.t = 1$, u will atomically add $v.t$ by 1 (line 32) and the propagation of v can run again with $v.A_p$ avoiding repeated propagation (line 16).

At the termination of the inner for-loop by $w.t = 0$, we have $w.V = w.adj$ so that the invariant of the while-loop holds. At the termination of the outer while-loop, the propagation queue $R = \emptyset$, so that all vertices $v \in V$ have $v.mcd$ correctly maintained as in Equation 1.

B.2 Complexities

Time Complexity. When m' edges are removed from the graph, the total work is the same as the sequential version in Algorithm 2, which is $O(m'|E^*|)$ where E^* is the largest number of adjacent edges for all vertices in V^* among each removed edge, defined as $E^* = \sum_{v \in V_p^*} v.deg$. Analogies to edge insertion, the best-case running time is $O(m'|E^*|/\mathcal{P} + |E^*| + m'|V^*|)$; the worst-case running

time is $O(m'|E^*|)$. In practice, such a worst-case is unlikely to happen.

Space Complexity. For each vertex $v \in V$, it takes $O(1)$ space to store $v.mcd$ and locks, which makes $O(n)$ space in total. Each worker p maintains a private V_p^*, which takes $O(|V^*|\mathcal{P})$ space in total. Each worker p maintains a private R_p, which takes $O(|E^*|\mathcal{P})$ space in total since at most $O(|E^*|)$ vertices can be added to R_p for each removed edge. The OM data structure is used to maintain the k-order for all vertices in the graph, which takes $O(n)$ space. Therefore, the total space complexity is $O(n + |V^*|\mathcal{P} + |E^*|\mathcal{P}) = O(n + |E^*|\mathcal{P})$.

C Experiments

C.1 Scalability Evaluation

In this experiment, we test the scalability over four selected graphs, e.g., *livej*, *baidu*, *dbpedia*, *roadNet-CA*. For each graph, we first randomly select from 100,000 to 1 million edges. By using 16 workers, we measure the accumulated running time and evaluate the ratio of real running time between the corresponding size of edges and 100,000 edges. The plots in Figure 6 depict the performance of four compared algorithms. The x-axis is the size of inserted or removed edges, and the y-axis is the time ratio. Ideally, 1 million edges should have a ratio of 10 since the edge size is 10 times of 100,000. We observe that over *livej*, four algorithms always have similar time ratios with increased edge size. Over other graphs, `OurI` and `OurR` always have larger time ratios compared with `JEI` and `JER`, respectively. Further, `OurI` has a time ratio of up to 20 when applying 1 million edges. This is because `JEI` or `JER` adopts the joint edge set structure to preprocess a batch of updated edges; if there are more updated edges, they can process more edges in each iteration and avoid unnecessary access. However, `OurI` and `OurR` do not preprocess a batch of updated edges so more updated edges require more accumulated running time.

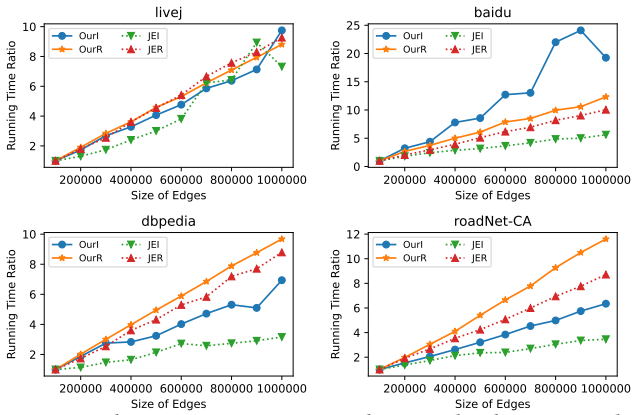

Figure 6: The running time ratio with 16-worker by varying the size of inserted or removed edges.

We also observe that even with 1 million update edges, `OurI` and `OurR` still have better performance than `JEI` and `JER`, respectively. Over four tested graphs, `OurI` still has 2.6x, 1.9x, 3.8x and 3.0x speedups compared with `JEI`, and `OuR` also has 7.8x, 5.5x, 0.9x and 3.3x speedups compared with `JER`, respectively. The reason is that `OurI` and `OurR` (based on the Order algorithm) have less work than `JEI` and `JER` (based on the Traversal algorithm; also, unlike `OurI` and `OurR`, `JEI` and `JER` have an extra cost to preprocess the edges.

C.2 Stability Evaluation

In this experiment, we test the stability over four selected graphs, e.g., *livej*, *baidu*, *dbpedia*, *roadNet-CA*, by using 16 workers. First, we randomly sample 5, 000, 000 edges and partition them into 50 groups, where each group has totally different 100, 000 edges. Second, for each group, we measure the accumulated running time of different methods. That is, the experiments run 50 times so each time it has totally different inserted or removed edges.

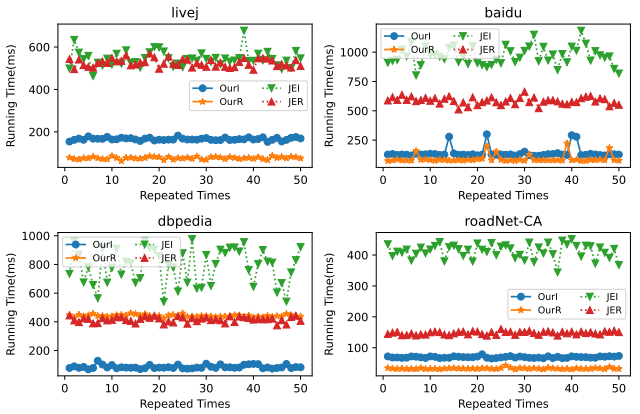

Figure 7: The running time with 16-worker by varying a batch of inserted or removed edges for each time.

The plots in Figure 7 depict the result. The x-axis is the repeated times, and the y-axis is the running times. We observe that the performance of `OurI`, `OurR`, and `JER` are always well-bounded, but the performance of `JEI` always has large fluctuations. The reason is that `JEI` is based on the Traversal algorithm and `OurI` is based on the Order algorithm. It is proved that for the edge insertion, the Traversal algorithm has large fluctuations for the ratio of $|V^+|/|V^*|$ for different edges with high probability, while the Order algorithm does not have this problem. For the edge removal, both `Our` and `JER` have $V^+ = V^*$, so their running times remain stable for different batches of edges.