Parallel Order-Based Core Maintenance in Dynamic Graphs

Bin Guo
McMaster University
Hamilton, Canada
guob15@mcmaster.ca

Emil Sekerinski
McMaster University
Hamilton, Canada
emil@mcmaster.ca

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. Our 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:

Bin Guo and Emil Sekerinski. 2023. Parallel Order-Based Core Maintenance in Dynamic Graphs. In *52nd International Conference on Parallel Processing (ICPP 2023), August 07–10, 2023, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3605573.3605597

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 vertices 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 [31], gene expression [8], social networks [10], ecology [3], and finance [3].

In [22], 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, 24, 32]. 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 [9, 23, 25]. This is a problem of maintaining the core number in dynamic graphs. In [36], Zhang, et al. summarize the research on core maintenance and applications.

Many sequential algorithms are devised on core maintenance in dynamic graphs [12, 18, 26, 27, 34, 37]. 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 scope of vertices (denoted as V^+). There are two main algorithms, Order [37] and Traversal [27]. 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 comes 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, 30]. 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 [18, 26], which is much less efficient than the Order algorithm [12, 13]. Second, they exploit the parallelism only for vertices with different core numbers, that is, they reduce to sequential algorithms when all affected vertices have the same core numbers.

ICPP 2023, August 07–10, 2023, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0843-5/23/08…$15.00
https://doi.org/10.1145/3605573.3605597

To overcome the above drawbacks, inspired by the Simplified-Order algorithm [12], we propose a new parallel algorithm to maintain core numbers 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 the total amount of computations performed by the algorithm and the *depth* is the longest chain of sequential dependencies [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 will block as a chain and then reduce to sequential execution. Fewer locked vertices will lead to higher parallelism.
- When inserting edges in parallel, we lock affected vertices in order to avoid deadlocks. When removing edges in parallel, 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 evaluating the performance of different algorithms.

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 [32], 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 personal computer over large graphs by using `GraphChi` and `WebGraph` models. In [24], 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 shared memory parallel algorithm is reported in [15].

Core Maintenance. In [18, 26], an algorithm that is similar to the Traversal algorithm is given, but this solution has quadratic time complexity. In [32], a semi-external algorithm for core maintenance is proposed in order to reduce the I/O cost, but this method is not optimized for CPU time. In [28], Sun et al. design algorithms to maintain approximate cores in dynamic *hypergraphs* in which a *hyperedge* 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 [35], 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 [33], the distributed approaches to core maintenance are explored. In [21], the parallel approximate k-core decomposition and maintenance approach is proposed, where bounded approximate core numbers for vertices can be maintained with high probability.

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 [38], 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 [12, 37]. 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 induced subgraph G_k of G is called a k-core if it satisfies: (1) for $\forall u \in V(G_k)$, $u(G_k).deg \geq k$, 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 contains vertex u is denoted as 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 linear time $O(m + n)$ algorithm, so-called BZ algorithm, shown in Algorithm 1. The core number of u is determined in line 5. The min-priority queue Q can be efficiently implemented by bucket sorting [1], leading to a linear running time of $O(m + n)$.

Algorithm 1: BZ algorithm for core decomposition

```
1 for u ∈ V do  u.d ← |u.adj|; u.core = ∅
2 Q ← a min-priority queue by u.d for all u ∈ V
3 while Q ≠ ∅ do
4     u ← Q.dequeue()
5     u.core ← u.d; remove u from G
6     for v ∈ u.adj do
7         if u.d < v.d then v.d ← v.d − 1
8     update Q
```

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^*, we have to traverse a possibly larger set of vertices, denoted as V^+.

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 [26] insertion algorithm. The Order removal algorithm has $V^+ = V^*$ and has to maintain the k-order of vertices $\mathbb{O}$, which is also faster than Traversal removal algorithm. This is why we try to parallelize the Order insertion and removal algorithms in this paper.

In [18, 26], it is proved that after inserting or removing one edge, the core number of vertices in V^* increase or decrease at most one, respectively. Also, when inserting an edge (u, v), we can search for V^* in the k-subcore, where k is the lower core numbers between u and v.

3.1 The Order-Based Core Maintenance

The state-of-the-art core maintenance solution is the Order algorithm [12, 37]. 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 [26].

Edge Insertion.

Definition 3.5 (k-Order $\preceq$)**.** [37] 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 BZ algorithm. When generating the k-order, there may be multiple vertices $v \in Q$ that have the same value of $u.d$ and can be dequeued from Q at the same time together (Algorithm 1, line 4). When dealing with these vertices with the same value of d, different sequences generate different instances of correct k-order for all vertices. There are three heuristic strategies, "small degree first", "large degree first", and "random". The experiments in [37] show that the "small degree first" consistently has the best performance over all tested real graphs, and thus we choose this strategy for implementation and experiments.

For the k-order, transitivity holds, that is, $u \preceq v$ if $u \preceq w \wedge w \preceq v$. For each edge insertion and removal, the k-order will 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 the 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, 37] 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, 37] 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 $v.d^+_{out} = |\{w \in v.post : w \notin V^+ \setminus V^*\}|$.

Theorem 3.1. [12] *Given a constructed DAG $\vec{G} = (V, \vec{E})$ by inserting an edge $u \mapsto v$ with $K = u.core \leq v.core$, the candidate set V^* includes all possible vertices that satisfy: 1) their core numbers equal to K, and 2) their total numbers of candidate in-degree and remaining out-degree are greater than K, denoted as $\forall w \in V : w \in V^* \equiv (w.core = K \ \wedge \ w.d^*_{in} + w.d^+_{out} > K)$*

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 after adding v to V^*. Theorem 3.1 shows what qualified vertices should be added into V^*. In this case, V^* and V^+ are maintained, which is used to calculate $v.d^*_{in}$ and $v.d^+_{out}$ when traversing v.

Edge Removal.

Definition 3.8 (max-core degree *mcd*)**.** [12, 27, 37] 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 $v.mcd = |\{w \in v.adj : w.core \geq v.core\}|$.

All vertices v in G maintain $v.mcd \geq v.core$. When removing an edge (u, v), e.g. $v.core < u.core$, we have $v.mcd$ off by 1 and $u.mcd$ unchanged. In this case, if $v.mcd < v.core$, Edge removal maintenance is required.

3.2 Order Maintenance Data Structure

In the Simplified-Order core maintenance algorithm [12], the sequential Order Maintenance (OM) data structure [2, 7] is used to maintain the k-order. The OM data structure has the following three operations:

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

Specifically, 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 [29] is used. That is, each item is stored in a *bottom-list*; each group is stored in a *top-list*, which can contain $\Omega(\log N)$ items. Both the top-list and the bottom-list are organized as double-linked lists. 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. The `Order`(x, y) operation can determine if x precedes y by comparing the labels, denoted as $x \leq y \equiv L^t(x) < L^t(y) \ \vee \ (L^t(x) = L^t(y) \wedge L_b(x) < L_b(y))$, which requires $O(1)$ time.

For `Insert`(x, y) operation, 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 `Insert`(x, y) operation costs amortized $O(1)$ time as the relabel process costs amortized $O(1)$ time for each `Insert` operation. Additionally, the `Delete`(x) operation can directly remove x without affecting the labels by $O(1)$ time. Based on the above sequential OM data structure, a parallel version of the OM data structure is designed in [11].

In this work, we adopt the parallel OM data structure [11] to maintain the k-order in parallel for three advantages. First, our method has a larger portion of `Order` operations to compare the order of two items compared with `Insert` and `Delete` operations. The lock-free `Order` operations are efficient even if multiple works are inserting or removing vertices at the same time. Second, all three operations cost $O(1)$ works, which will not scarify the work complexity of our core maintenance. Third, the labels of vertices, which indicate their order, and can be used to implement the priority queue Q. Here, Q is the key data structure for our core maintenance in Algorithm 5.

3.3 Atomic Primitive and Lock

The compare-and-swap atomic primitive `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`.

Algorithm 2: Lock(x) with c

1 **while** c **do**
2   **if** x = false $\wedge$ `CAS`(x, false, true) **then**
3   **if** c **then return** true
4   **else** $x \leftarrow$ false; **return** false
5 **return** false

We implement a condition-lock as in Algorithm 2. 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.

4 PARALLEL CORE MAINTENANCE

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 insertion are shown in Algorithm 3. Given an undirected graph G, the core numbers and k-order for all vertices are 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, $\Delta E_1 \ldots \Delta E_{\mathcal{P}}$, 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 `InsertEdge`$_p$ (line 4). The key issue is how to implement `InsertEdge`$_p$ executed by a worker p in parallel with other workers.

Removing edges in parallel is analogous to Algorithm 3, and the key issue is `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, 30], prepossessing of ΔE_p is not required so that edges can be inserted or removed on-the-fly.

4.1 Parallel Edge Insertion

Algorithm. The detailed steps of `InsertEdge`$_p$ are shown in Algorithm 5. 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 they 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

Algorithm 3: Parallel-InsertEdges($G, \mathbb{O}, \Delta E$)

1 partition ΔE into $\Delta E_1, \ldots, \Delta E_{\mathcal{P}}$
2 `DoInsert`$_1(\Delta E_1) \parallel \cdots \parallel$ `DoInsert`$_{\mathcal{P}}(\Delta E_{\mathcal{P}})$
3 **procedure** `DoInsert`$_p(\Delta E_p)$
4   **for** $(u, v) \in \Delta E_p$ **do** `InsertEdge`$_p(G, \mathbb{O}, (u, v))$

Algorithm 4: Parallel-Order($\mathbb{O}, u, v$)

1 $s \leftarrow \emptyset; s' \leftarrow \emptyset; r \leftarrow \emptyset$
2 **do**
3   **do** $s \leftarrow u.s; s' \leftarrow v.s$ **while** $s \bmod 2 = 1 \lor s' \bmod 2 = 1$
4     $r \leftarrow u \preceq v$
5 **while** $s \neq u.s \lor s' \neq v.s$
6 **return** r

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 4, when comparing the order of u and v, we ensure that u and v are not updating their k-order. We repeatedly acquire $u.s$ and $v.s$ as s and s' until both s and s' are even numbers (line 3). After comparing the order of u and v (line 4), we check if $u.s$ and $v.s$ are increased or not (line 5). If that is the case, we redo the whole process (line 2). Finally, we return the result in line 6.

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 $u.d^+_{out} \leq K$, 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 $w.d^*_{in} + w.d^+_{out} \leq K \land w.d^*_{in} > 0$, vertex w does $\texttt{Backward}_p$ (line 11). If $w.d^*_{in} + w.d^+_{out} \leq K \land w.d^*_{in} = 0$, 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 Q_p is empty (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 in Algorithm 5 are almost the same as their sequential version since all vertices in V^+ are locked. There are a few 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. In Figure 1, we show an example of maintaining the core numbers of vertices in parallel after inserting three edges. 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 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

Algorithm 5: InsertEdge$_p(\vec{G}, \mathbb{O}, u \mapsto v)$

1 Lock u and v together when both are not locked
2 **if** $v \preceq u$ **then** $\texttt{Unlock}$ u and v; **goto** line 1
3 $V^*_p, V^+_p, K, \leftarrow \emptyset, \emptyset, \texttt{min}(u.core, v.core)$
4 insert $u \mapsto v$ into $\vec{G}$ with $u.d^+_{out} \leftarrow u.d^+_{out} + 1$
5 $\texttt{Unlock}(v)$
6 **if** $u.d^+_{out} \leq K$ **then** $\texttt{Unlock}(u)$; **return**
7 $Q_p, w \leftarrow$ a min-priority queue by $\mathbb{O}, u$
8 **do**
9     $w.d^*_{in} \leftarrow |\{w' \in w.pre : w' \in V^*_p\}|$ // calculate d^*_{in}
10   **if** $w.d^*_{in} + w.d^+_{out} > K$ **then** $\texttt{Forward}_p(w)$
11   **else if** $w.d^*_{in} > 0$ **then** $\texttt{Backward}_p(w)$ **else** $\texttt{Unlock}(w)$
12     $w \leftarrow Q_p.dequeue()$ with $w.core = K$ and $\texttt{Lock}(w)$
13 **while** $w \neq \emptyset$
14 **for** $w \in V^*_p$ **do**
15     $w.core \leftarrow K + 1$; $w.d^*_{in} \leftarrow 0$
    // atomically add $w.s$
16     $\langle w.s{+}{+}\rangle$; $\texttt{Delete}(\mathbb{O}_K, w)$; $\texttt{Insert}(\mathbb{O}_{K+1}, head, w)$; $\langle w.s{+}{+}\rangle$
17 Unlock all locked vertices
18 **procedure** $\texttt{Forward}_p(u)$
19     $V^*_p \leftarrow V^*_p \cup \{u\}$; $V^+_p \leftarrow V^+_p \cup \{u\}$ // u is locked
20   **for** $v \in u.post : v.core = K$ **do**
21   **if** $v \notin Q_p$ **then** $Q_p.enqueue(v)$
22 **procedure** $\texttt{Backward}_p(w)$
23     $V^+_p \leftarrow V^+_p \cup \{w\}$; $pre \leftarrow w$ // w is locked
24     $R_p \leftarrow$ an empty queue; $\texttt{DoPre}_p(w, R_p)$
25     $w.d^+_{out} \leftarrow w.d^+_{out} + w.d^*_{in}$; $w.d^*_{in} \leftarrow 0$
26   **while** $R_p \neq \emptyset$ **do**
27         $u \leftarrow R_p.dequeue()$
28         $V^*_p \leftarrow V^*_p \setminus \{u\}$
29         $\texttt{DoPre}_p(u, R_p)$; $\texttt{DoPost}_p(u, R_p)$
        // atomically add $w.s$
30         $\langle w.s{+}{+}\rangle$; $\texttt{Delete}(\mathbb{O}_K, u)$; $\texttt{Insert}(\mathbb{O}_K, pre, u)$; $\langle w.s{+}{+}\rangle$
31         $pre \leftarrow u$; $u.d^+_{out} \leftarrow u.d^+_{out} + u.d^*_{in}$; $u.d^*_{in} \leftarrow 0$
32 **procedure** $\texttt{DoPre}_p(u, R_p)$
33   **for** $v \in u.pre : v \in V^*_p$ **do**
34         $v.d^+_{out} \leftarrow v.d^+_{out} - 1$
35   **if** $v.d^*_{in} + v.d^+_{out} \leq K \land v \notin R_p$ **then** $R_p.enqueue(v)$
36 **procedure** $\texttt{DoPost}_p(u, R_p)$
37   **for** $v \in u.post$ **do**
38   **if** $v \in V^*_p \land v.d^*_{in} > 0$ **then**
39             $v.d^*_{in} \leftarrow v.d^*_{in} - 1$
40   **if** $v.d^*_{in} + v.d^+_{out} \leq K \land v \notin R_p$ **then** $R_p.enqueue(v)$

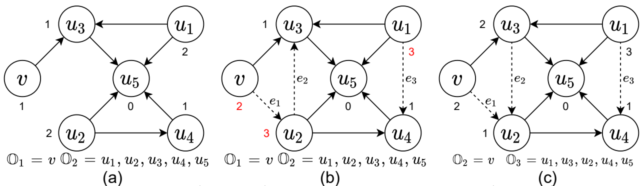

Figure 1: An example graph maintaining core numbers after inserting edges, e_1, e_2, and e_3. The letters inside the circle are vertices' IDs and $\mathbb{O}_k$ is the k-order of vertices with core numbers k. The number beside each vertex is its remaining out-degree d^+_{out}. The direction for each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

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).

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.

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 5).
- 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. For the details of implementation, please refer to Appendix.

Correctness. We only argue the correctness of Algorithm 5 related to the concurrent part, as the correctness of its sequential version has been argued in [12]. There are no deadlocks by lock vertices in order. For each worker p, the accessed vertices are synchronized by locking. When locking u, the sets $u.post$ and $u.pre$ will not change until u is unlocked. Please refer to the arxiv version[1] for full proof.

Time and Space Complexities. The best-case running time is $O(m'|E^+|\log|E^+|/\mathcal{P} + |E^+|\log|E^+| + m'|V^*|)$, and the worst-case running time is $O(m'|E^+|\log|E^+|)$, where m' edges are inserted and E^+ is the largest number of adjacent edges for all vertices in V^+ among each inserted edge. The worst-case is unlikely to happen in practice. The total space is $O(n + |E^+|\mathcal{P})$. Refer to the arxiv version for full proof.

4.2 Parallel Edge Removal

Algorithm. The detailed steps of $\texttt{RemoveEdge}_p$ are shown in Algorithm 6. 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).

[1] https://arxiv.org/pdf/2210.14290.pdf

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.

Example 4.2. In Figure 2, we show an example of maintaining the core numbers of vertices in parallel when removing three edges. 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(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.

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.

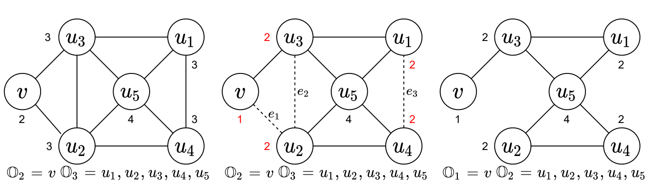

Figure 2: An example graph maintains the core numbers after removing 3 edges, e_1, e_2, and e_3. The letters inside the cycles are vertices' IDs and the $\mathbb{O}_k$ is the k-order of vertices with core numbers k. The beside numbers are corresponding mcd. (a) an initial example graph. (b) remove three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

Algorithm 6: $\text{RemoveEdge}_p(G, \mathbb{O}, (u, v))$

```
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
```

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 into $u_3.mcd$ 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. There have no deadlocks because 1) both u and v are locked together for a removed edge (u, v) (line 1), and 2) for all vertices $w \in R_p$, we have w locked by the worker p and $w.core = K - 1$ and also worker p will lock all $w' \in w.adj$ with $w.core = K$ for propagation. For all vertices v, their $v.mcd$ are correctly maintained without locking the neighbors $v.adj$. Please refer to the arxiv version for full proof.

Time and Space Complexities. The best-case running time is $O(m'|E^*|/\mathcal{P} + |E^*| + m'|V^*|)$ and the worst-case running time is $O(m'|E^*|)$, where m' edges are removed and E^* is the largest number of adjacent edges for all vertices in V^* among each removed edge. The worst-case is unlikely to happen in practice. The total space is $O(n+|E^*|\mathcal{P})$. Please refer to the arxiv version for full proof.

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) [27]

The source code is available on GitHub[2].

The experiments are performed on a server with an AMD Ryzen Threadripper 3990X (64 cores 2.9 GHz, 128 hyperthreads, 256 MB of L3 Cache) and 256 GB of main memory. Each CPU core corresponds to a worker. The server runs 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[3] 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.

5.1 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[4]. The *dbpedia*, *baidu*, *pokec* and *wiki-talk-en wiki-links-en* graphs are collected from the KONECT[5] project. The *ER*, *BA*, and *RMAT* graphs are synthetic graphs generated by the SNAP[6] 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 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 the latest 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, where the "AvgDeg" is the average degree and "Max k" is the maximal core numbers among all vertices.

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. 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; all vertices in *roadNet-CA* have four core numbers from 0 to 3; all vertices in *BA* have a single core number as 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.

5.2 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 continuous 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

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

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.

The numbers of locked vertices evaluation, the speedup evaluation, the scalability evaluation, and the stability evaluation are in the arxiv version.

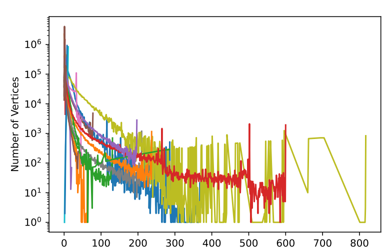

Figure 3: The vertices' core number distributions, where the x-axis is core numbers and the y-axis is the number of vertices with the same core number. It shows the trend for all graphs and thus the legend is omitted.

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. 2003. An $O(m)$ Algorithm for Cores Decomposition of Networks. *CoRR* cs.DS/0310049 (2003). http://arxiv.org/abs/cs/0310049

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

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

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

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

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

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

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

[9] Kasimir Gabert, Ali Pinar, and Ümit V Çatalyürek. 2021. Shared-Memory Scalable k-core Maintenance on Dynamic Graphs and Hypergraphs. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 998–1007.

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

[11] Bin Guo and Emil Sekerinski. 2022. New Parallel Order Maintenance Data Structure. *arXiv preprint arXiv:2208.07800* (2022).

[12] Bin Guo and Emil Sekerinski. 2022. 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. 2019. Faster parallel core maintenance algorithms in dynamic graphs. *IEEE Transactions on Parallel and Distributed Systems* 31, 6 (2019), 1287–1300.

[14] Hai Jin, Na Wang, Dongxiao Yu, Qiang Sheng Hua, Xuanhua Shi, and Xia Xie. 2018. Core Maintenance in Dynamic Graphs: A Parallel Approach Based on Matching. *IEEE Transactions on Parallel and Distributed Systems* 29, 11 (nov 2018), 2416–2428. https://doi.org/10.1109/TPDS.2018.2835441 arXiv:1703.03900

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

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

[17] Yi-Xiu Kong, Gui-Yuan Shi, Rui-Jie Wu, and Yi-Cheng Zhang. 2019. k-core: Theories and applications. *Physics Reports* 832 (2019), 1–32. https://doi.org/10.1016/j.physrep.2019.10.004

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

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

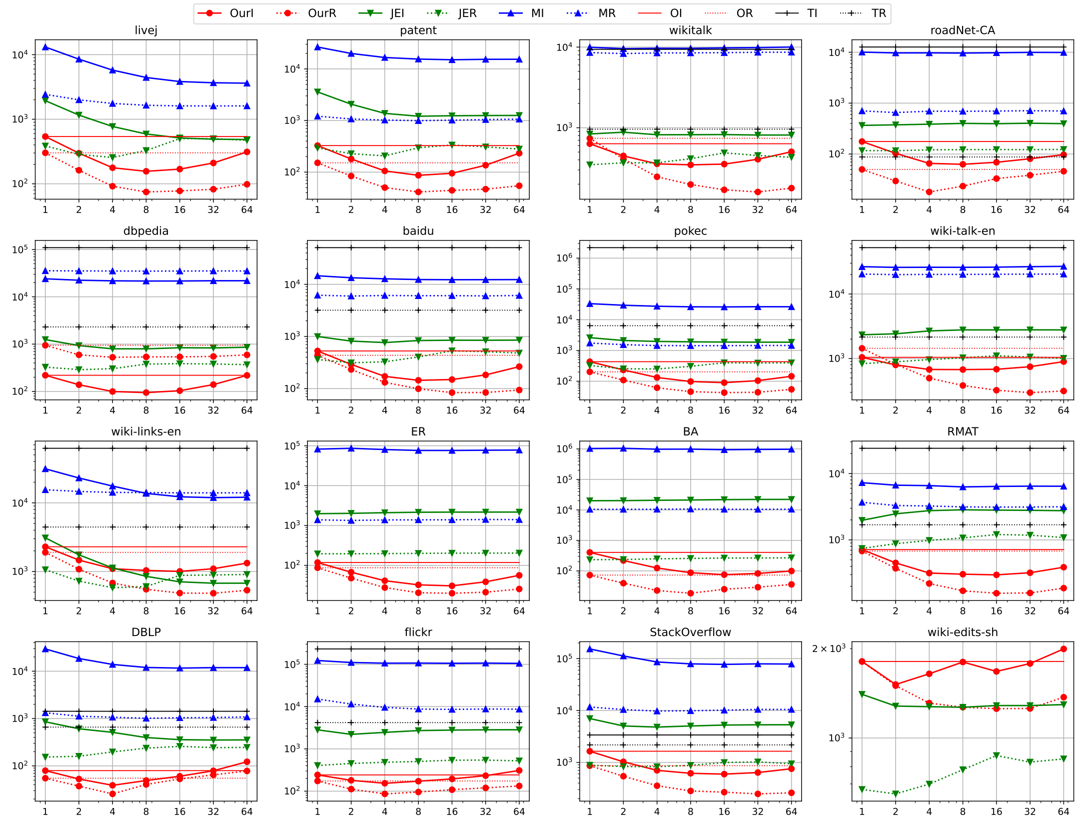

Figure 4: The real running time by varying the number of workers. The x-axis is the number of workers and the y-axis is running time (millisecond).

[20] Bin Liu and Feiteng Zhang. 2020. Incremental Algorithms of the Core Maintenance Problem on Edge-Weighted Graphs. *IEEE Access* PP (04 2020), 1–1. https://doi.org/10.1109/ACCESS.2020.2985327

[21] Quanquan C Liu, Jessica Shi, Shangdi Yu, Laxman Dhulipala, and Julian Shun. 2021. Parallel Batch-Dynamic Algorithms for k-Core Decomposition and Related Graph Problems. *In Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '22)* (2021).

[22] Fragkiskos D Malliaros, Christos Giatsidis, Apostolos N Papadopoulos, and Michalis Vazirgiannis. 2020. The core decomposition of networks: Theory, algorithms and applications. *The VLDB Journal* 29, 1 (2020), 61–92. https://doi.org/10.1007/s00778-019-00587-4

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

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

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

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

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

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

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

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

[31] Ping Wang, Xingdong Deng, Yang Liu, Liang Guo, Jun Zhu, Lin Fu, Yakun Xie, Weilian Li, and Jianbo Lai. 2022. A Knowledge Discovery Method for Landslide Monitoring Based on k-core Decomposition and the Louvain Algorithm. *ISPRS International Journal of Geo-Information* 11, 4 (2022), 217.

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

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

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

[35] Michael Yu, Dong Wen, Lu Qin, Ying Zhang, Wenjie Zhang, and Xuemin Lin. 2021. On querying historical k-cores. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2033–2045.

[36] Feiteng Zhang, Bin Liu, and Qizhi Fang. 2020. Core Decomposition, Maintenance and Applications. In *Complexity and Approximation*. Springer, 205–218.

[37] Yikai Zhang, Jeffrey Xu Yu, Ying Zhang, and Lu Qin. 2017. A fast order-based approach for core maintenance. In *Proceedings - International Conference on Data Engineering*. 337–348. https://doi.org/10.1109/ICDE.2017.93 arXiv:1606.00200

[38] Wei Zhou, Hong Huang, Qiang-Sheng Hua, Dongxiao Yu, Hai Jin, and Xiaoming Fu. 2021. Core decomposition and maintenance in weighted graph. *World Wide Web* 24, 2 (2021), 541–561.