Parallel Order-Based Core Maintenance in Dynamic Graphs

Bin Guo, Emil Sekerinski

***Abstract*—The core number of vertices in a graph is one of the most well-studied cohesive subgraph models due to linear running time algorithms. In practice, many data graphs are dynamic graphs that continuously change by inserting and removing edges. The core numbers are updated in dynamic graphs with edge insertions and deletions, called core maintenance. When a burst of a large number of inserted or removed edges comes in, these must be handled on time to keep up with the data stream. There are two main sequential algorithms for core maintenance, TRAVERSAL and ORDER. Experiments show that the ORDER algorithm significantly outperforms the TRAVERSAL algorithm over various 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. This paper proposes a new parallel core maintenance algorithm based on the ORDER algorithm. A distinguishing properly is that the algorithm allows parallelism even for graphs where all vertices have the same core number. 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, the proposed method achieves speeds of up to 289 times and 10 times compared to the most efficient existing method.

***Index Terms*—dynamic graphs, parallel, k-core maintenance, multicore, shared memory.**

I. INTRODUCTION

GRaphs are widely used to model complex networks. As one of the most well-studied cohesive subgraph models, the k-core is the maximal subgraph such that all vertices have degrees at least k. The *core number* of a vertex is the maximum value of k such that this vertex is contained in the subgraph of k-core vertices [1], [2]. The core numbers can be computed in 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 is a parameter of density extensively used in numerous applications [2], such as knowledge discovery [3], gene expression [4], social networks [5], ecology [6], and finance [6].

In [7], Malliaros et al. summarize the main research on k-core decomposition from 1968 to 2019. Many papers focus on computing the core in static graphs [1], [8]–[11]. In practice, many data graphs are large and continuously changing. It is of practical importance to identify the dense range as fast as possible after a change, e.g., when 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 [12]–[14]. This is the problem of maintaining the core number in dynamic graphs. In [15], Zhang et al. summarize the research on core maintenance and applications.

Many sequential algorithms are devised for core maintenance in dynamic graphs [16]–[21]. The main idea for core maintenance is that first, a set of vertices whose core numbers need to be updated (denoted as V^*) is identified by traversing a possibly larger scope of vertices (denoted as V^+) such that $V^* \subseteq V^+$. There are two main algorithms for maintaining core numbers over dynamic graphs, TRAVERSAL [18] and ORDER [17]. Given an inserted edge, the TRAVERSAL algorithm searches V^* by performing a depth-first graph traversal within a *subcore*, a connected region of vertices with the same core numbers. For the ORDER algorithm, the size of V^+ is significantly reduced, so the ratio $|V^+|/|V^*|$ is typically much smaller and has less variation compared to the TRAVERSAL algorithm. Thus, the computational time is significantly improved. The experiments in [17] show that for edge insertion, ORDER significantly outperforms TRAVERSAL over all tested graphs with up to 2,083 times speedups; for the edge removal, ORDER outperforms TRAVERSAL over most of the tested graphs with up to 11 times speedups. Furthermore, based on the ORDER algorithm, a SIMPLIFIED-ORDER algorithm is proposed in [16], which is easier to implement and to argue for correctness; also, SIMPLIFIED-ORDER has improved time complexities by adopting the *Order Maintenance* (OM) data structure to maintain the order of all vertices.

All the above methods are sequential for maintaining core over dynamic graphs, meaning they handle only one edge insertion or removal at a time. The problem is that when a burst of many inserted or removed edges comes in, these edges may not be handled on time to keep up with the data stream [14]. 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 [22]–[24]. These algorithms are based on similar ideas: 1) they use an available structure, e.g. *Join Edge Set* [22], to preprocess a batch of inserted or removed edges, avoiding repeated computations, and 2) each worker performs the TRAVERSAL algorithm. However, there are three drawbacks to these approaches. First, they are based on the sequential TRAVERSAL algorithm [20], [21], which is proved to be less efficient than the ORDER algorithm [16], [22]. Second, parallelism can be exploited only for affected vertices with different core numbers; the algorithm is reduced to running sequentially when all affected vertices have the same core numbers. Third, the time complexities are not analyzed by

Bin Guo is with the Department of Computing and Information Systems, Trent University, Peterborough, ON, Canada

Emil Sekerinski is with the Department of Computing and Software, McMaster University, Hamilton, ON, Canada

TABLE I
THE WORK AND DEPTH COMPLEXITIES OF OUR PARALLEL CORE MAINTENANCE

	Worst-case (O)		Best-case (O)	
Parallel	$\mathcal{W}$	$\mathcal{D}$	$\mathcal{W}$	$\mathcal{D}$
Insert	$m'\|E^+\|\log\|E^+\|$	$m'\|E^+\|\log\|E^+\|$	$m'\|E^+\|\log\|E^+\|$	$\|E^+\|\log\|E^+\| + m'\|V^*\|$
Remove	$m'\|E^*\|$	$m'\|E^*\|$	$m'\|E^*\|$	$\|E^*\| + m'\|V^*\|$

the *work-depth* model [25], where the work $\mathcal{W}$ is its sequential running time, and the depth $\mathcal{D}$ is its running time on an infinite number of processors.

To overcome the above drawbacks, inspired by the SIMPLIFIED-ORDER algorithm [16], we propose a new parallel batch algorithm, called PARALLEL-ORDER, to maintain core numbers after batches of edges insertions or removals in dynamic graphs. Based on maintaining the k-order with the parallel OM data structure, we can parallelize the ORDER algorithm. 1) For edge insertion, the idea is straightforward: each worker handles one inserted edge at a time and propagates the affected vertices in k-order, such that only the traversed vertices in V^+ are locked. When traversing $v \in V^+$, not all neighbors $u \in v.adj$ need to be locked if $u \notin V^+$. When another worker tries to access these vertices, it must wait until they are unlocked. Since all affected vertices are locked in k order, this method does not have blocking cycles that lead to a deadlock. 2) Edge removal is more challenging than edge insertion. The idea is that each worker handles one removed edge at a time and propagates the affected vertices, such that only the traversed vertices in V^* are locked. When traversing $v \in V^*$, not all neighbors $u \in v.adj$ need to be locked if $u \notin V^*$. The problem is that this method may cause blocking cycles that lead to a deadlock as all affected vertices are traversed without any order. We propose a novel mechanism to avoid such deadlocks. The main contributions of this work are summarized below:

- We investigate the drawbacks of the state-of-the-art parallel core maintenance algorithms [22]–[24].
- Based on the parallel OM data structure [26] and the SIMPLIFIED-ORDER core maintenance algorithm [16], we propose a PARALLEL-ORDER edge insertion algorithm for core maintenance. Only the traversed vertices in V^+ are locked for synchronization. We also implement the priority queue Q combined with the parallel OM data structure to obtain a vertex in Q with a minimal k-order.
- We propose a PARALLEL-ORDER edge removal algorithm for core maintenance and a novel mechanism to avoid blocking cycles that lead to a deadlock. Only the traversed vertices in V^* are locked for synchronization.
- We prove the correctness of our PARALLEL-ORDER insertion and removal core maintenance algorithms; we also prove the time complexities with the work-depth model.
- We conduct extensive experiments on a 64-core machine over various graphs to evaluate the PARALLEL-ORDER algorithms for edge insertion and removal.

We analyze our parallel algorithms in the standard *work-depth* model [27]. The *work*, denoted as $\mathcal{W}$, is the total number of operations that the algorithm uses. The *depth*, denoted as $\mathcal{D}$, is the longest chain of sequential operations. Table I shows the work and depth of the PARALLEL-ORDER algorithm for inserting and removing m' edges in parallel. For both edge insertion and removal, one issue is that the depth $\mathcal{D}$ is equal to the work $\mathcal{W}$ in the worst case; that is, all workers execute in one blocking chain such that only one worker is active. However, with a high probability, such a worst-case does not happen as the number of locked vertices is always small for each insertion and removal. For our method, all vertices in V^+ or V^* are locked together for each insertion or removal. In Fig. 1, on 16 tested graphs (in Table II of our experiment section), we summarize the number of different sizes of V^+ when randomly inserting and removing 100,000 edges. We observe that almost all V^+ and V^* have really small sizes for insertion or removal, respectively. Specifically, more than 97% of edge insertions and removals for all tested graphs have $|V^+|$ and $|V^*|$ between 0 and 10. That means, with a high probability that less than 11 vertices are locked when inserting or removing one edge, which leads to a low probability that all workers execute in one long blocking chain.

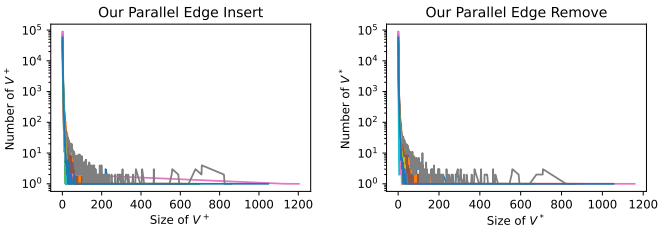

Fig. 1. The number of different sizes of V^+ for inserting and removing 100,000 edges by using our parallel core maintenance algorithms, respectively. The x-axis is the size of V^+ and the y-axis is the number of such size of V^*.

The rest of this paper is organized as follows. The related work is discussed in Section II. The preliminaries are given in Section III. Our new parallelized Order-Based core maintenance algorithms are proposed in Section IV. We discuss the implementation of priority queues and buffer queues in Section V. We conduct extensive performance studies and show the results in Section VI, and conclude this work in Section VII.

II. RELATED WORK

A. Core Decomposition

The BZ algorithm [1] has linear running time $O(m)$ by using bucket structures, where m is the number of edges. In [8], 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 [11],

Wen et al. provide a semi-external algorithm, which requires $O(n)$ memory to maintain the information of vertices. In [9], Khaouid et al. investigate the core decomposition in a single PC over large graphs by using `GraphChi` and `WebGraph` models. In [10], 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 [28], where the ParK algorithm was proposed. Based on the main idea of ParK, a more scalable PKC algorithm has been reported in [29].

B. Core Maintenance

In [20], [21], an algorithm similar to the TRAVERSAL algorithm is given, but with a quadratic time complexity. In [11], a semi-external algorithm for core maintenance is proposed to reduce the I/O cost, but this method is not optimized for CPU time. In [30], 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 [14], Gabert et al. propose parallel core maintenance algorithms for maintaining cores over hypergraphs. There exists some research based on core maintenance. In [31], the authors study computing all k-cores in the graph snapshot over the time window. In [32], the authors explore the hierarchy core maintenance. In [33], distributed approaches to core maintenance are explored. In [34], a parallel approximate k-core decomposition and maintenance approach is proposed, where bounded approximate core numbers for vertices can be maintained with high probability.

C. Weighted Graphs

All the above work focuses on unweighted graphs, but graphs are weighted in many applications. For an edge-weighted graph, the degree of a vertex is the sum of the weights of all its incident edges. However, weighted graphs have a large search range for maintaining the core numbers after a change by using the traditional core maintenance algorithms directly, as the degree of a related vertex may change widely. In [35], 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 [16], [17]. In [36], Liu et al. improve the core decomposition and incremental maintenance algorithm to suit edge-weighted graphs.

III. PRELIMINARIES

Let $G = (V, E)$ be an undirected and unweighted graph; $V(G)$ 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 $E(G)$, 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. We define the set of neighbors of a vertex $u \in V$ as $u.adj$, formally $u.adj = \{v \in V : (u, v) \in E\}$. We denote the degree of u in G as $u.deg = |u.adj|$.

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

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
```

Definition III.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 III.3 (k-Subcore)**.** Given an 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 `sc`(u).

A. 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, the so-called BZ algorithm, shown in Algorithm 1. The general idea is *peeling*: to compute the k-core G_k of G, repeatedly vertices (and their adjacent edges) whose degrees are less than k are removed. When there are no more vertices to be removed, the resulting graphs are the k-core of G. 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 $O(m + n)$.

B. Core Maintenance

The problem of maintaining the core numbers for dynamic graphs G when edges are inserted into and removed from G continuously is called core maintenance. The insertion and removal of vertices can be simulated as a sequence of edge insertions and removals.

Definition III.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 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 V^+, is traversed by accessing their adjacent edges.

Clearly, we have $V^* \subseteq V^+$. An efficient core maintenance algorithm should have a small ratio of $|V^+|/|V^*|$. It is shown that the ORDER [22] insertion algorithm has a significantly

lower ratio compared to the TRAVERSAL [20] insertion algorithm. This is why we try to parallelize the ORDER algorithm in this paper.

In [20], [21], it is proved that after inserting or removing one edge, the core number of vertices in V^* increases or decreases by at most one, respectively; V^* is only located in the k-subcore, where k is the lower core number of two vertices that the inserted or removed edge connect.

C. The Order-Based Core Maintenance

The state-of-the-art core maintenance solution is the ORDER algorithm [16], [17]. For edge insertion, it is based on three notions: k-order, candidate degree, and remaining degree. For edge removal, it uses the notion of the max-core degree [20].

1) Edge Insertion:

Definition III.5 (k-Order $\preceq$)**.** [17] 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 $u.core < v.core$, we have $u \preceq v$; (2) when $u.core = v.core$, we have $u \preceq v$ if u's core number is determined before v's by the peeling steps of the BZ algorithm.

A k-order $\preceq$ is an instance of all the possible vertex sequences produced by the 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 [17] 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 set of successors of v is defined as $u(\vec{G}).post = \{v \mid u \mapsto v \in \vec{E}(\vec{G})\}$; the set of 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 [16].

Definition III.6 (candidate in-degree)**.** [16], [17] Given a constructed DAG $\vec{G}(V, \vec{E})$, the candidate in-degree $v.d^*_{in}$ is the total number of its predecessors located in V^*, denoted as $d^*_{in}(v) = |\{w \in v.pre : w \in V^*\}|$.

Algorithm 2: InsertEdge($\vec{G}, \mathbb{O}, u \mapsto v$)

```
1  V*, V+, K ← ∅, ∅, u.core
2  insert u ↦ v into G with u.d+_out ← u.d+_out + 1
3  if u.d+_out ≤ K then return
4  Q ← a min-priority queue by ℙ; 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
   ℙ_K and insert w at the beginning of ℙ_{K+1} in k-order.
```

Definition III.7 (remaining out-degree)**.** [16], [17] Given a constructed DAG $\vec{G}(V, \vec{E})$, the remaining out-degree $v.d^+_{out}$ is the total number of its 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 III.1. [16] 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, formally $\forall w \in V : w \in V^* \equiv (w.core = K \ \wedge \ w.d^*_{in} + w.d^+_{out} > K)$

For all vertices v in $\vec{G}$, we must ensure that $v.core \leq v.d^+_{out}$. When inserting an edge $v \mapsto u$, the out-degree $v.d^+_{out}$ increases by 1. If $v.core > v.d^+_{out}$, edge insertion maintenance is required after adding v to V^*. Theorem III.1 shows what qualified vertices should be added to V^*. In this case, V^* and V^+ are maintained, which are used to calculate $v.d^*_{in}$ and $v.d^+_{out}$ when traversing v.

Algorithm 2 is the ORDER insertion algorithm [16]. As the precondition, we assume that $v.core$, $v.d^+_{out}$, and $v.d^*_{in}$ for all vertices v are correctly initialized. 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 to 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 one 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 must not be added to V^*; procedure `Backward` 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 $\mathbb{O}$ not in the above two cases are skipped. Finally, V^* includes all vertices whose core numbers should be added by 1 (line 9). Of course, the k-order is maintained for inserting other edges (line 10).

Example III.1. Fig. 2 shows an example of maintaining the

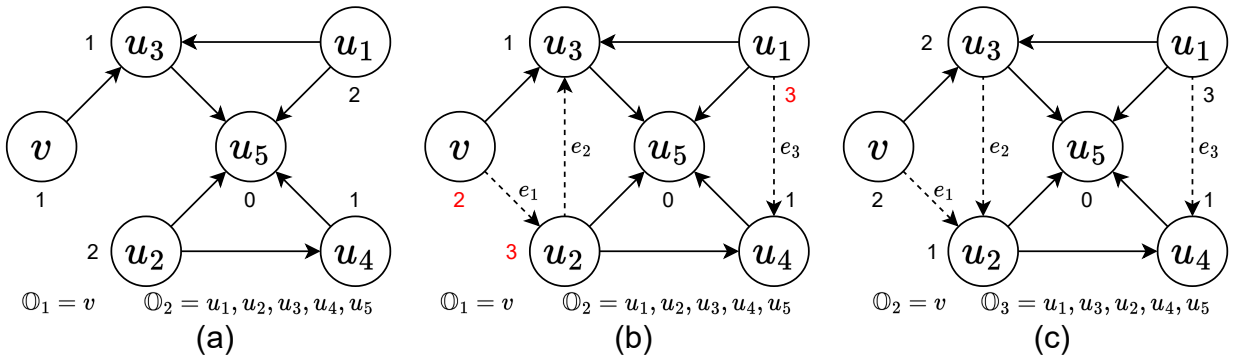

Fig. 2. An example graph maintains the core numbers after inserting three edges, e_1, e_2, and e_3 . The letters inside the cycles are vertices' IDs and the $\mathbb{O}_k$ is the k-order of vertices with core numbers k. The beside numbers are corresponding remaining out-degrees d^+_{out}. The direction for each edge indicates the k-order of two vertices, which is constructed as a DAG. (a) an initial example graph. (b) insert 3 edges. (c) the core numbers and k-orders update.

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

Figure 2(b) shows three edges, e_1, e_2 and e_3, being 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 removed from V^* (by `Backward`). In this case, u_2 is moved after u_1 in 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 can be 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 2(c) shows the result after inserting edges. All vertices have core numbers increased by 1. Orders $\mathbb{O}_2$ and $\mathbb{O}_3$ are updated accordingly. All vertices' d^+_{out} are updated accordingly.

2) Edge Removal:

Definition III.8 (max-core degree mcd)**.** [16]–[18] 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.

The ORDER removal algorithm is presented in Algorithm 3. After an edge is removed, the affected vertices, u and v, have to be put into V^* if their mcd less than $core$ (lines 2 to 4), which may repeatedly cause the other vertices' mcd to decrease and then be 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 next time (line 11). Also, all vertices' mcd have to be updated for removing an edge next time (line 12).

Algorithm 3: RemoveEdge($G, \mathbb{O}, (u, v)$)

1 $R, K, V^* \leftarrow$ an empty queue, $\texttt{Min}(u.core, v.core), \emptyset$
2 remove (u, v) from $\vec{G}$ with updating $u.mcd$ and $v.mcd$
3 **if** $u.mcd < K$ **then** $V^* \leftarrow V^* \cup \{u\}$; $R.enqueue(u)$
4 **if** $v.mcd < K$ **then** $V^* \leftarrow V^* \cup \{v\}$; $R.enqueue(v)$
5 **while** $R \neq \emptyset$ **do**
6 $w \leftarrow R.dequeue()$
7 **for** $w' \in w.adj$ **with** $w'.core = K \wedge w \notin V^*$ **do**
8 $w'.mcd \leftarrow w'.mcd - 1$
9 **if** $w'.mcd < K$ **then**
  $V^* \leftarrow V^* \cup \{w'\}$; $R.enqueue(w')$
10 **for** $w \in V^*$ **do** $w.core \leftarrow w.core - 1$
11 Remove all $w \in V^*$ from $\mathbb{O}_K$ and append to $\mathbb{O}_{K-1}$ in k-order
12 update mcd for all related vertices accordingly

Example III.2. Fig. 3 shows an example of maintaining the core numbers of vertices after successively removing three edges, e_1 to e_3. Figure 3(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 $\mathbb{O}_3$. For all vertices, the core numbers are less than or equal to mcd.

Fig. 3(b) shows the three edges, e_1 to e_3, removed. (1) For e_1, $v.mcd$ is off by 1 so we have $v.mcd < K = 2$ and $V^* = \{v\}$, but $u_2.mcd$ is not affected. There is no propagation since all $v.adj$ have core numbers greater 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 to R for propagation, and u_1, u_4, and u_5 are consecutively added to V^* with $V^* = \{u_2, u_3, u_1, u_4, u5\}$. Finally, we decrease the core number of u_1 to u_5 from 3 to 2; also, the mcd of both u_2 and u_3 are updated as 2, and the mcd of u_1, u_4 and u_5 are updated as 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 stops. Finally, the mcd of both u_1 and u_4 are updated to 2.

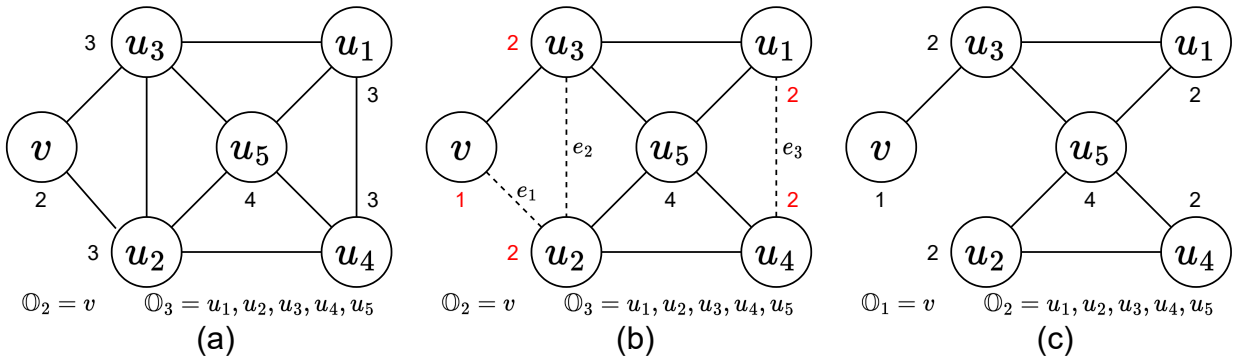

Fig. 3. An example graph maintains the core numbers after removing 3 edges, e_1, e_2, and e_3. The letters inside the cycles are vertices' IDs and the $\mathbb{O}_k$ is the k-order of vertices with core numbers k. The beside numbers are corresponding mcd. (a) an initial example graph. (b) remove three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

Figure 3(c) shows the result after removing the edges. All vertices have core numbers that decreased by 1. Orders $\mathbb{O}_1$ and $\mathbb{O}_2$ are updated accordingly. Also, all vertices' mcd are updated accordingly.

D. Order Maintenance Data Structure

In the SIMPLIFIED-ORDER core maintenance algorithm [16], the OM data structure [37], [38] is used to maintain the k-order. In this work, we adopt a concurrent version of the OM data structure [26] to maintain the k-order for the parallel core maintenance. 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 there are maximal N items in the total order $\mathbb{O}$. All items are assigned labels to indicate the order. For the `Insert` operation, a *two-level* data structure [39] is used. Each item is stored in the bottom-list, which contains a *group* of consecutive elements; each group is stored in the top-list, which can contain $\Omega(\log N)$ items. Both the top-list and bottom-list are organized as double-linked lists. We use $x.pre$ and $x.next$ to denote the predecessor and successor of x, respectively. 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. With 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 is 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 the group g of x 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.

Such rebalance and split operations will continue until less than $\frac{\log N}{2}$ items remain in g. In addition, new bottom labels L_b are uniformly assigned for items in g.

Label L^t is in the range $[0, N^2]$ and label L_b in the range $[0, N]$. Typically, each label can be stored as an $O(\log N)$ bits integer. Assume it takes $O(1)$ time to compare two integers. For the sequential version [37]–[39], each `Insert` operation only cost amortized $O(1)$ time; also, the `Order` and `Delete` operations requires $O(1)$ time. In the parallel version [26], `Insert` and `Delete` are synchronized by locks. More importantly, the parallel `Order` is lock-free, which is meaningful since a large portion of OM operations for core maintenance in graphs is to compare the order of two vertices on one edge.

In this work, we adopt the parallel OM data structure [26] to maintain the k-order in parallel for three reasons. First, our method has a larger portion of `Order` operations compared to `Insert` and `Delete` operations. The lock-free `Order` operations are efficient even if multiple workers insert and remove vertices concurrently. Second, all three operations cost $O(1)$ work, which does not worsen the work complexity of our core maintenance. Third, the labels of vertices, which indicate their order, can be used to implement the priority queue Q. Here, Q is the key data structure for our core maintenance in Algorithm 7.

E. 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.

We use locks for synchronization in our parallel algorithms. In our experiments, we implemented two kinds of locks. One is the locks of OpenMP [40], `omp_set_lock` and `omp_unset_lock`, which is easy to use, but the overhead

is high. In this paper, we use the compare-and-swap primitive (CAS) [41]. We assume that the locks are somewhat fair.

Given variable x as a lock, the CAS will repeatedly check x, and set x from false to true if x is false. One worker will busy-wait on lock x without suspension until another worker releases the lock.

Additionally, we use the conditional lock as in Algorithm 4. The condition c is checked before and after the CAS primitive (lines 1 and 3). It is possible that other workers may update the condition c concurrently. If c is changed to false after locking x, variable x will be unlocked and then return false immediately (line 4). The conditional Lock can atomically lock x by satisfying condition c, thus avoiding blocking on a locked x that does not satisfy c.

Algorithm 4: Lock(x) with c

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

IV. Parallel Core Maintenance

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

The steps for parallel inserting edges are shown in Algorithm 5. Given an undirected graph G, the core number and k-order can be initialized by the BZ algorithm [1] in linear time. A batch of edges ΔE are to be inserted in 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 handles multiple edges in ΔE_p in parallel with other workers (line 2). Each time, 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 worker p in parallel with other workers.

Algorithm 5: 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))
```

Removing edges in parallel is analogous to Algorithm 5; the key issue is $\texttt{RemoveEdge}_p$. Note that the 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 [22]–[24], a prepossessing of ΔE_p is not required so that edges can be inserted and removed on-the-fly.

A. Parallel Edge Insertion

1) Algorithm: The detailed steps of $\texttt{InsertEdge}_p$ are shown in Algorithm 7, which is analogous to Algorithm 2. We introduce several new data structures. First, the priority queue Q_p, the queue R_p, the candidate set V_p^*, and the searching set V_p^+ are all privately used by the worker p; 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 to 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 6, 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 $v.s$ have increased (line 5). In that case, we redo the whole process (line 2). Finally, we return the result in line 6.

Algorithm 6: Parallel-Order($\mathbb{O}, u, v$)

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

Given an edge $u \mapsto v$ to be inserted, where $u \preceq v$, we lock both u and v together at the same time when both are not locked (line 1). We redo the lock of u and v if they were updated by other workers as $v \preceq u$ (line 2). After locking, K is initialized as the smaller core number of u and v. After inserting the edge $u \mapsto v$ in 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 to u for propagation (line 7). In the do-while-loop (lines 8 - 13), initially, w equals u, which is already locked in line 1 (line 7). We calculate $w.d_{in}^*$ by counting the number of $w.pre$ located in V_p^* while w is locked since w may be accessed by other workers (line 9). If $w.d_{in}^* + w.d_{out}^+ > K$, vertex w requires the $\texttt{Forward}$ procedure_p (line 10). If $w.d_{in}^* + w.d_{out}^+ \leq K \wedge w.d_{in}^* > 0$, vertex w requires the Backward procedure (line 11). If $w.d_{in}^* + w.d_{out}^+ \leq K \wedge w.d_{in}^* = 0$, we can skip w and unlock w since w can not be in V^+ (line 11). Repeatedly, we dequeue w from Q_p (line 12) with the following steps: 1) we lock the head w of Q_p; 2) by checking $w.s$, we know whether w has been locked and updated by other workers or not; 3) if that is the case, we remove w from Q_p if $w.core > K$, unlock w and update Q_p. In a word, we dequeue w from Q_p, where w has the minimal k-order in Q_p and $w.core = K$ (line 12). After locking w, its k-order cannot be changed by other workers. The do-while-loop terminates when no vertices can be dequeued from Q_p (line 13). All vertices $w \in V_p^*$ have core numbers incremented by 1 and their $w.d_{in}^*$ is reset to 0 (line

15); also, all vertices 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 $w.s$ is atomically incremented by 1 before and after this process (line 16). Before termination, all locked vertices w are unlocked (line 17).

Procedures `Forward`(u) and `Backward`(w) are almost the same as their sequential version since all vertices in V^+ are locked. There are only several differences. In the `Forward`$_p(u)$ procedure, for each v in $u.post$ whose core number equals to K, we add v in 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 using. In procedure `Backward`$_p(w)$, all vertices w are removed from $\mathbb{O}_K$ and appended after pre to maintain the k-order by using the parallel OM data structure, where $w.s$ is atomically incremented by 1 before and after this process (line 30).

Example IV.1. Continuing in Fig. 2, we show an example of maintaining the core numbers of vertices in parallel after inserting three edges. Figure 2(b) shows the insertion of three edges, e_1, e_2 and e_3 in parallel by three workers, p_1, p_2, and p_3, respectively. (1) For e_1, worker p_1 will first lock v and u_2 for inserting the edge. But u_2 is already locked by p_2, so p_1 has to wait for p_2 to finish and unlock u_2. (2) For e_2, worker p_2 will first lock 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 have that V^* is empty. Continuing, 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, worker p_3 first locks 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 unlocked by p_2) with $u_3.d^*_{in} = 1$, so u_3 is added to V^* and u_2 is added to Q_3 for propagation. Continuing, 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 unlocked by p_2).

We can see how three vertices, u_3, u_4 and u_5, can be added to 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.

2) Correctness: We only argue the correctness of Algorithm 2 related concurrency. 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 established as $w = u$ and $Q_p = \emptyset$.

Algorithm 7: InsertEdge$_p(\vec{G}, \mathbb{O}, u \mapsto v)$

```
1  Lock u and v together if 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)
       // w is locked and w.core = K when dequeuing
          w from Q_p
12     w ← Q_p.dequeue(K)
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)
```

When dequeuing w from Q, worker p first locks w, which has the minimum in the k-order, and then removes 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; 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 procedure `Backward`; also, all vertices v' are moved after v in k-order, they cannot be moved before v. In other words, all vertices in Q_p cannot have a smaller k-order than w when w is locked.

Worker p traverse $u.post$ in the for-loop (lines 20 - 21, 37 - 40), where u is locked by p; but, not all $u.post$ are locked by p and may be locked by other workers for updating. So are $u.pre$ 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 vertices v 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 set of $u.post$ and $u.pre$ will not change until u is unlocked, even when other workers access the vertices in $u.post$ and $u.pre$.

3) Time Complexity: When m' edges are inserted into a graph, the total work is the same as that of the sequential version in Algorithm 2, 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, k 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 total work, 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.

4) 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 take $O(3n)$ 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

1) Algorithm: The steps of $\texttt{RemoveEdge}_p$ are shown in Algorithm 8. We introduce several new data structures. First, the queue R_p is privately used by worker p, which cannot be accessed by other workers and synchronization is not necessary (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 $v.t$, which has four possible values:

- $v.t = 2$ means $v.core$ is off by 1 add will be propagated by adding to R. Note that, the $v.core = K-1$ and $v.t = 2$ have to be executed atomically (line 22), as two values indicate one status.
- $v.t = 1$ means v is being propagated by the inner for-loop (lines 11 - 14).
- $v.t = 3$ means v has to be checked again since some vertices $v.adj$ have core numbers decreased by other workers.
- $v.t = 0$ means v is just initialized or already propagated.

Given a removed edge (u, v), we lock both u and v together at the same time when both are not locked (line 1). After locking, K is initialized to the smaller core numbers of u and v (line 2). We execute procedure $\texttt{CheckMCD}_p$ for u or v to ensure $u.mcd$ and $v.mcd$ are not 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 than or equal to K, we execute procedure $\texttt{DoMCD}_p$ (lines 5 and 6), by which u and v may be added to R_p for propagation. If u or v are 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 conditionally locked with $w'.core = K$ (line 11 and 12). Note that, to avoid deadlock, when locking w', we will not busy-wait once the condition changed to $w'core \neq K$, as $w'.core$ can be decreased from K to $K-1$ by other workers (line 12). For each locked $w' \in w.adj$, we first execute procedure $\texttt{CheckMCD}_p$ in case $w'.mcd$ is empty (line 15) and then execute the $\texttt{DoMCD}_p$ procedure (line 13); also, the visited w' are added to A_p to avoid visiting them repeatedly. We atomically decrease $w.t$ by 1 before and after the 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). All locked vertices are unlocked before termination (line 18).

In procedure $\texttt{DoMCD}_p(u)$, vertex u has been locked by worker p (line 19). We decrease $u.mcd$ by 1, and $u.mcd$ 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 (line 22) since $v.t$ indicates v's status for other workers. 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; it can be calculated later if used (line 24).

In procedure $\texttt{CheckMCD}(u)$, we recalculate $u.mcd$ if it is empty (line 27). We initially set a temporary mcd to 0 (line 28), and then count the $u.mcd$ (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 that is the case, we increment the temporary mcd by one (line 30). When $v.core = K-1$, it is possible that $v.t$ is being updated by other workers. If $v.t$ equals 1, we know that v is being propagated. In this case, we have to set $v.s$ from 1 to 3 by the atomic primitive `CAS`, which leads to the propagation of v to be redone in line 16 by other workers (line 32). 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 notable advantage is that we calculate $u.mcd$ without locking $v \in u.adj$.

Algorithm 8: RemoveEdge$_p(G, \mathbb{O}, (u, v))$

```
1  Lock u and v together if 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
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 IV.2. Continuing in Fig. 3, we show an example of maintaining the core numbers of vertices in parallel when removing three edges. Figure 3(b) shows the removal of three edges, e_1, e_2 and e_3 in parallel by three workers, p_1, p_2 and p_3, respectively. (1) For e_1, worker p_1 locsk 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 $u_2.mcd$; the core number of v is off by 1 and added to R_1 for propagation. Since only one vertex, $u_3 \in v.adj$, has a core number greater than v, the propagation of v terminates. Finally, v is unlocked. (2) For e_2, 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 vertices $u_2.adj$; vertex u_4 is locked by worker p_3. At the same time, $u_4.core$ is decreased from 2 to 1, and p_1 skips to lock 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 vertices $u_3.adj$ 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 first locks u_1 and u_4 together for removing the edge. Then both $u_1.core$ and $u_4.core$ are off by 1, and 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 that p_2 and p_3 execute without blocking each other, and all vertices in V^* are locked.

The above example assumes that all vertices' mcd are initially generated. If $u_3.mcd = \varnothing$ 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 propagated u_3, we have to force u_1 to redo the propagation (setting $u_1.t$ from 1 to 3 atomically).

2) Correctness: Algorithm 8 has no deadlocks. First, both u and v are locked together for a removed edge (u, v) (line 1). Second, all vertices $w \in R_p$ are locked by 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 vertices w' are not locked, there is no deadlocks;
- if w' is locked by another worker q but $w'.core$ is not decreased, there is no deadlock 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 is no deadlock as w' is skipped for traversing.
- importantly, if w' is locked by another worker q and $w'.core$ is decreased from K to $K - 1$, there is no deadlock 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 ensures to stop busy-waiting when $w'.core$ decreases from K to $K - 1$.

The key issue of Algorithm 8 is to correctly maintain the mcd of all vertices in the graph:

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

With this definition of mcd, 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 that v is locked and $\neg v.lock$ to denote v is unlocked. We use R to denote the union of all propagation queues, $R = \cup_{p=1}^{\mathcal{P}} R_p$. A vertex $v \in R$ indicates v is in one of the R_p for worker p.

The invariant of this while-loop is that all vertices $w \in V$ maintain a status $w.t$ indicating if w in R or not; for all vertices $w \in R_p$, which are locked and added to V^*, their core numbers are off by 1 and mcd set as to empty (waiting to be recalculated). 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:

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

The invariant initially holds as vertices u and v may be added to R_p to remove the edge (u, v), and u and v are locked if added to R_p. We now argue that the while-loop preserves 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, and $w.s$ is off to 0 when w is removed from R_p for proportion.
- $\forall w \in R_p : w.core = K - 1 \wedge w.mcd = \varnothing \wedge w \in V_p^* \wedge w.lock \wedge w.s > 0$ 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 being added to 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 procedure `CheckMCD`, u may have neighbors $v \in u.adj$ whose core numbers are off by 1 and are 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 is empty 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 being 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 follows. 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 the $w.V$ to empty (line 16). We use V^* to denote the union of all V_p^*, formally $V^* = \cup_{p=1}^{\mathcal{P}} V_p^*$. A vertex $v \in V^*$ indicates v is in one of the V_p^* for worker p.

The invariant of the outer while-loop (lines 8 - 16) 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. The status $v.t = 1$ indicates that v is doing the propagation and $v.t = 0$ indicates that v has finished the propagation:

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

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 off by 1 before and after the for-loop; also, $w.t$ may be atomically incremented 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 procedure `CheckMCD`, vertex 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 has already been removed from R before the inner for-loop (line 9) for propagation; there are three cases:
  1) 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 $u.mcd$.
  2) if v has already traverse u such that $v.core = u.core - 1$ for propagation as $u \in v.A_p$, u should not count v as $u.mcd$.
  3) if v has already traverse 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 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 $v.t = 1$, vertex u will atomically increment $v.t$ 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 the invariant of the while-loop holds.

At the termination of the outer while-loop, the propagation queue R is empty, so that all vertices $v \in V$ have $v.mcd$ correctly maintained as in Equation IV-B2.

3) Time Complexity: When m' edges are removed from the graph, the total work is the same as the sequential version in Algorithm 3, 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 $E^* = \sum_{v \in V_p^*} v.deg$. Analogous 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.

4) Space Complexity: For each vertex $v \in V$, it takes $O(1)$ space to store $v.mcd$ and locks, so the space in total is $O(n)$. Each worker p maintains a private set V_p^*, which takes $O(|V^*|\mathcal{P})$ space in total. Each worker p maintains a private set 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 $O(n + |V^*|\mathcal{P} + |E^*|\mathcal{P}) = O(n + |E^*|\mathcal{P})$.

V. Implementation

The min-priority queue Q is used in core maintenance for edge insertion to efficiently obtain a vertex $v \in Q$ with a minimum k-order $\mathbb{O}$, where $\mathbb{O}$ is maintained by the parallel OM data structure. Queue Q is implemented with a min-heap for comparing the labels maintained by the parallel OM data structure, which supports enqueuing and dequeuing in $O(\log |Q|)$ time.

The key issue is how to efficiently implement the enqueue and dequeue operations, when the labels of vertices in $\mathbb{O}_k$ are updated by the relable procedure (including rebalance and split) in the OM data structure. The solution is to re-make the min-heap of Q each time when the relable procedure is triggered in $\mathbb{O}_k$. For the implementation, we require the following structures:

- Each k-order $\mathbb{O}_k$ maintains a version number $\mathbb{O}_k.ver$, which is atomically incremented by 1 before and after one triggered relable procedure.
- Each k-order $\mathbb{O}_k$ maintains a counter $\mathbb{O}_k.cnt$, which can atomically record how many workers are executing the triggered relable procedure.
- The vertex v is added to Q along with the current top-label (group label), bottom-label, status and version number, denoted as $[L_b(v), L^t(v), v.s, ver]$. These two labels are used for min-priority in Q.
- All vertices $v \in Q$ should have the same version number, which equals to Q's version number $Q.ver$.

Definition V.1 (Version Invariant)**.** All vertices v in Q maintain the invariant that all $v.ver$ are the same version numbers as $Q.ver$, denoted as $\forall u, v \in Q : u.ver = v.ver = Q.ver$.

The `dequeue` operation preserves the Version Invariant for all vertices v in Q, as an inconsistent version number of vertices may lead to wrong results. The steps of updating $Q.ver$ are shown in Algorithm 9. Initially, we set ver' as the current version of $\mathbb{O}_K$ (line 1). If $ver' \neq Q.ver$, all vertices v in Q will have their $[L_b(v), L^t(v), v.s, ver']$ updated to the current new values, with ver' as their version numbers (lines 4 - 7). We have to ensure that $\mathbb{O}_k.cnt = 0$ and $ver' = \mathbb{O}_k.ver$ during such updating; otherwise, we will redo the updating (lines 2 and 8). We also have to ensure that $v.s$ is an even number and is not changed during updating; otherwise, we have to redo the updating (lines 5 and 7) since other workers have accessed the vertices in Q and their k-order may be changed. In other words, no other workers can execute during the updating. Finally, we set $Q.ver$ to ver' since all versions in Q have the same version as ver' (line 6).

Algorithm 9: $Q.update\_version(\mathbb{O}_k)$

```
1 ver' ← O_K.ver
2 if O_k.cnt ≠ 0 ∨ ver' ≠ O_K.ver then goto line 1
3 if ver' ≠ Q.ver then
4 |   for v ∈ Q do
5 |   |   s' ← v.s
6 |   |   Update v with current [L_b(v), L^t(v), s', ver']
7 |   |   if ¬Even(s') ∨ s' ≠ v.s then goto line 5
8 if O_K.cnt ≠ 0 ∨ ver' ≠ O_K.ver then goto line 1
9 Q.ver ← ver'
```

1) Enqueue: The details steps of the `enqueue` operation are shown in Algorithm 10. We set ver' to the current version of $\mathbb{O}_K$ (line 1). For the vertex v, we add the values of $[L_b(v), L^t(v), v.s, ver']$ to Q, with ver' as their version numbers (line 2), and then update Q. If ver' is not consistent with $\mathbb{O}_k.ver$ or $Q_p.ver$, we set $Q_p.ver$ to $\varnothing$, which indicates the delayed version updating when executing `dequeue` operations.

Algorithm 10: $Q.enqueue(\mathbb{O}_k, v)$

```
1 ver', s' ← O_K.ver, v.s
2 Add v into Q with [L_b(v), L^t(v), v.s, ver'] and then update Q
3 if ver' ≠ O_K.ver ∨ ver' ≠ Q.ver ∨ s' ≠ s ∨ ¬Even(s) then
4 |   Q.ver ← ∅
```

2) Dequeue: Algorithm 11 shows the steps of the dequeue operation. If $Q.ver$ is empty, we update the version of Q so that all vertices in Q have consistent labels (line 2). In this case, we obtain v as $Q.front()$. which has the lowest k-order by comparing the labels (line 3). We conditionally lock v with $v.core = k$ as we will skip v when it has $v.core \neq k$ for the core maintenance (lines 4 and 5), since v can be accessed by other workers and has an increased core number. After locking v, it is necessary to check v's current status value $v.s$ with v's status value in Q (lines 6 and 7). If they are not equal, we know that v has been accessed by other workers, $v.core = k$, and v's k-order may be changed; then $Q.ver$ is set to empty to update the version in the next round (line 7). We remove v from Q and then return v, which is locked with the smallest k-order with $v.core = k$ (line 11). The whole process continues until we successfully obtain v from Q or Q is empty (lines 1

and 8). If no qualified v exists in Q, it will return empty (line 9).

Algorithm 11: $Q.dequeue(\mathbb{O}_k)$

```
1 while Q ≠ ∅ do
2     if Q.ver = ∅ then Q_p.update_version(O_k)
3     v ← Q.front()
4     if ¬(Lock v with v.core = k) then
5         Remove v from Q; continue
6     if v.s ≠ [v.s]_Q then
7         Unlock(v); Q_p.ver ← ∅; continue
8     Remove v from Q_p; return v
9 return ∅
```

3) Running time: The priority queue can be implemented by min-heap, which requires worst-case $O(\log |Q|)$ time for both enqueuing and dequeuing one item. For our implementation, with version updating, the priority queue requires worst-case $O(\log |Q)$ for enqueuing and $O(|Q| \log |Q|)$ for dequeuing, as we may rebuild the min-heap each time when removing a vertex. However, such a worst-case can happen with a low probability, as vertices are always inserted into the different positions of $\mathbb{O}_K$ and only a limited number of relable procedures can be triggered. Also, when inserting a batch of edges, the sizes of Q is always small, e.g. less than 10, so that the process of updating version tends to not affect the dequeue performance.

VI. Experiments

In this section, we experimentally compare our parallel core maintenance approach with the state-of-the-art *join-edge-set* core maintenance approach [22]. There are four algorithms:

- our parallel edge insertion algorithm (`OurI` for short) and removal algorithm (`OurR` for short),
- the *join edge set* based parallel edge insertion algorithm (`JEI` for short) and removal algorithm (`JER` for short)

A. 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. 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 [1] version 4.5 is used as the threading library. Our `OurI` and `OurR` use locks for synchronization, which are implemented by the `CAS` primitive for busy waiting. We perform every experiment at least 50 times and calculate their means with 95% confidence intervals.

B. Tested Graphs

We evaluate the performance of different methods over a variety of real-world and synthetic graphs, which are shown in Table II. For simplicity, directed graphs are converted to undirected ones in our testing; all of the self-loops and repeated edges are removed. That is, a vertex can not 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[2]. The *dbpedia*, *baidu*, *pokec* and *wiki-talk-en wiki-links-en* graphs are collected from the KONECT[3] project. The *ER*, *BA*, and *RMAT* graphs are synthetic graphs; they are generated by the SNAP[4] system using Erdös-Rényi, Barabasi-Albert, and the R-MAT graph models, respectively. For these generated graphs, 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, and we randomly sample edges for insertion and removal.

TABLE II
Tested real and synthetic graphs.

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

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

In Table II, we can see that all graphs have millions of edges, their average degrees ranges from 2.1 to 22.8, and their maximal core numbers ranges from 3 to 821. For most graphs, the core numbers are not well distributed. 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; *ER* has nine core numbers from 2 to 11; *BA* only has a single core number of 8. For `JEI` and `JER`, the core number distribution of graphs is an important property since the vertices with the same core number can only be handled by one worker at the same time, e.g., in *BA* only one worker can execute and all the other workers are wasted. However, `OurI` and `OurR` do not have such a limitation for parallelism, so all workers can always run in parallel over all tested graphs.

[1] https://www.openmp.org/

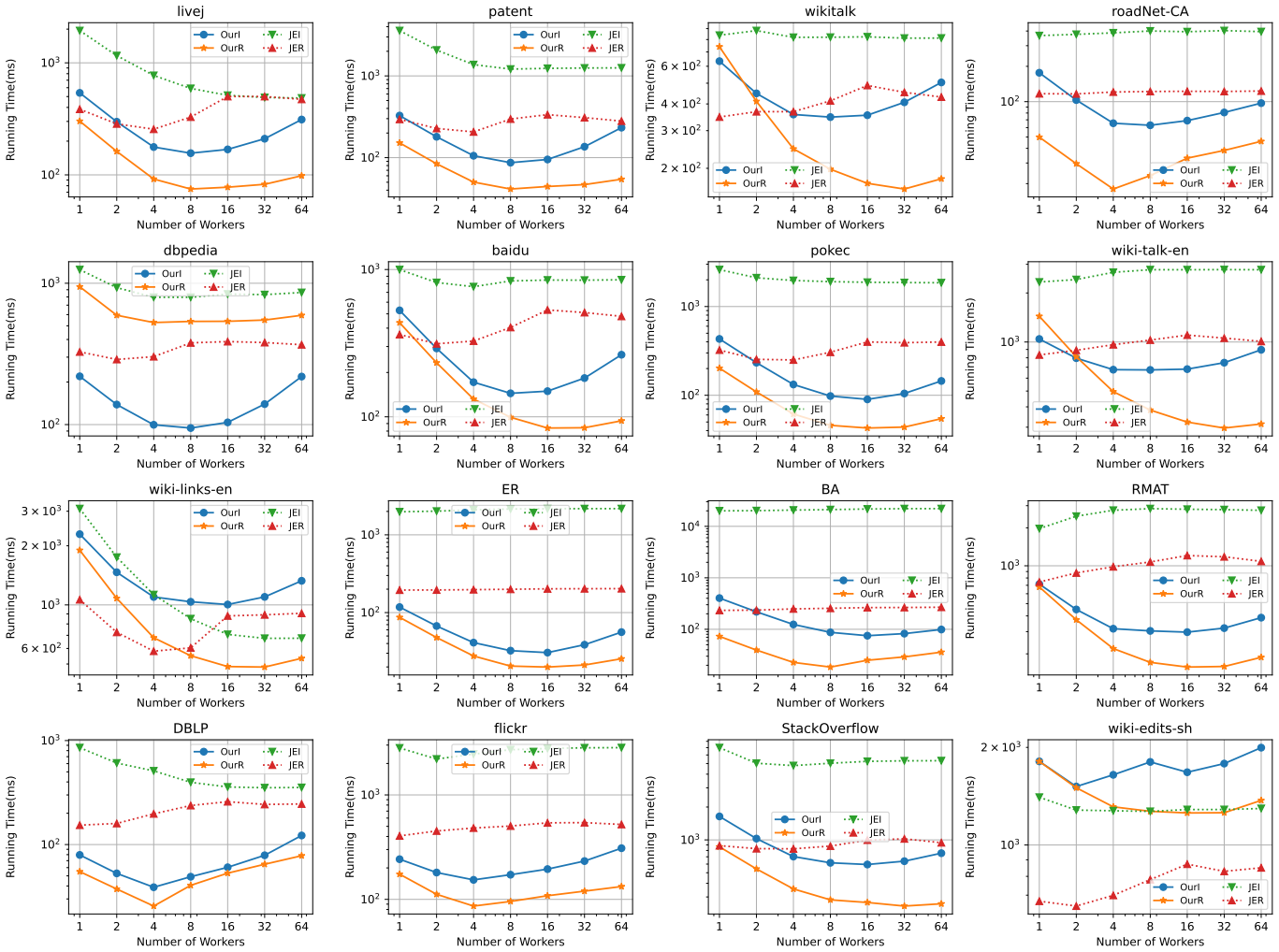

Fig. 4. The real running time by varying the number of workers. The x-axis is the number of workers, and the y-axis is the execution time (millisecond). The error bars are too small to show.

C. 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 II. For each graph, we first randomly select 100,000 edges. We measure the accumulated running time for inserting or removing such 100,000 edges. The plots In Fig. 4 depict the performance of the four compared algorithms. The first look over all tested graphs reveals that `OurI` and `OurR` always have better performance than `JEI` and `JER`, respectively. Specifically, we make several observations:

– By using one worker, all algorithms are reduced to running sequentially, and `OurI` performs much faster than `JEI`. This is because for edge insertion, `OurI` is based on the ORDER algorithm, while `JEI` is based on the TRAVERSAL algorithm. It is proved that the ORDER is much faster than TRAVERSAL for sequential version [17]. Also, `JEI` requires the preprocessing time to generate the join edge sets, while `OurI` can run on-the-fly without preprocessing.

– 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 needs to traverse $\log |u.adj|$ and $\log |v.adj|$ vertices. That means `OurR` has a higher running time than `JER` to delete an edge from the graph.

– By using multiple workers, `OurI` and `OurR` can always achieve better speedups compare with `JEI` and `JER`, respectively. This is because `JEI` and `JER` have limited parallelism, as affected vertices with different core numbers can not be processed in parallel, while `OurI` and `OurR` do not have such a limitation. On many real graphs, e.g., *wikitalk*, *roadNet-CA*, *ER*, *BA* and *RMAT*, the core numbers are not well-distributed, and a large percent of vertices have the same core numbers. Over such graphs, by increasing the number of workers, `JEI` and `JER` have

[2]http://snap.stanford.edu/data/index.html
[3]http://konect.cc/networks/
[4]http://snap.stanford.edu/snappy/doc/reference/generators.html

TABLE III
COMPARE THE SPEEDUPS.

Graph	1-worker vs 16-worker				1-worker		16-worker	
	OurI	OurR	JEI	JER	OurI vs JEI	OurR vs JER	OurI vs JEI	OurR vs JER
livej	3.2	3.9	3.8	0.8	3.6	1.3	3.0	6.5
patent	3.4	3.4	2.9	0.9	11.0	1.9	13.0	7.6
wikitalk	1.8	4.4	1.0	0.7	1.3	0.5	2.3	2.9
roadNet-CA	2.6	1.5	0.9	1.0	2.1	2.3	5.7	3.7
dbpedia	2.1	1.8	1.5	0.8	5.7	**0.3**	8.1	**0.7**
baidu	3.5	**5.2**	1.2	0.7	1.9	0.8	5.7	6.3
pokec	4.8	4.7	1.4	0.8	6.0	1.6	20.9	9.3
wiki-talk-en	1.5	4.5	0.8	0.8	2.2	0.6	4.1	3.4
wiki-links-en	2.3	3.9	**4.4**	**1.2**	1.3	0.6	**0.7**	1.8
ER	3.8	4.4	0.9	1.0	16.8	2.2	70.9	10.1
BA	**5.4**	2.9	0.9	0.9	**49.6**	**3.2**	**289.1**	**10.6**
RMAT	2.4	4.3	**0.7**	**0.6**	2.8	1.1	9.5	7.7
DBLP	1.3	**1.0**	2.4	0.6	10.8	2.8	5.9	4.9
flickr	1.2	1.6	1.0	0.7	11.6	2.3	14.3	5.0
StackOverflow	2.8	3.2	1.3	0.9	4.3	1.0	8.8	3.7
wiki-edits-sh	**1.1**	1.4	1.1	0.8	**0.8**	0.4	0.8	0.7

no speedups since one worker needs to traverse a large subgraph, which takes a great amount of time, but OurI and OurR can always obtain speedups since all workers have almost equal probability to traverse the graphs.

– The running time of all tested algorithms may begin to increase when using more than 8 or 16 workers in certain graphs, e.g., *livej*, *patent*, and *dbpedia*. This is because of the contention on the 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 III, columns 2 to 5 compare the running time speedups between using one worker and 16 workers for all tested algorithms. It is clear that OurI and OurR consistently achieve better speedups up to 5x, compared with JEI and JER. Columns 6 to 9 compare the running time speedups between our method and the compared method using one worker or 16 workers. We can see that compared with JEI, OurI achieves up to a 50x speedup even using one worker, and achieves up to a 289x speedup 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 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 our algorithms.

D. Scalability Evaluation

In this experiment, we test the scalability over four selected graphs, *livej*, *baidu*, *dbpedia*, *roadNet-CA*. For each graph, we first randomly select 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 Fig. 5 depict the performance of the four compared algorithms. 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*, the four

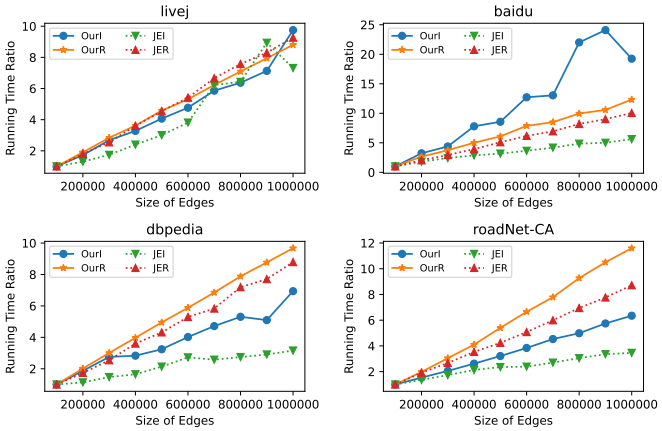

Fig. 5. The running time ratio with 16 workers by varying the size of the inserted or removed edges. The x-axis is the size of inserted or removed edges, and the y-axis is the time ratio.

algorithms always have similar time ratios with increased an edge size. Over other graphs, OurI and OurR always have larger time ratios compared to 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 edges require more accumulated running time.

We also observe that even with 1 million 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 to 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 extra cost to preprocess the edges.

E. Stability Evaluation

In this experiment, we test the stability over four selected graphs, *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, and each time has totally different inserted or removed edges.

The plots in Fig. 6 depict the result. We observe that the performance of `OurI`, `OurR`, and `JER` are always well-bounded, but the performance of `JEI` has larger 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 edge insertion, the TRAVERSAL algorithm has a large fluctuations ratio of $|V^+|/|V^*|$ for different edges with high probability, while the ORDER algorithm does not have this problem. For edge removal, both `OurR` and `JER` have $V^+ = V^*$, so their performance remains stable for different batches of edges.

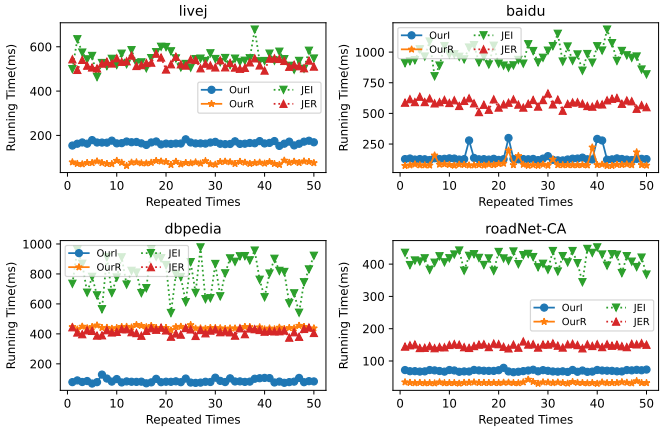

Fig. 6. The running time with 16-worker by varying a batch of inserted or removed edges for each time. The x-axis is the repeated times, and the y-axis is the running times.

VII. CONCLUSIONS AND FUTURE WORK

We present new parallel core maintenance algorithms for inserting and removing a batch of edges based on the ORDER algorithm. A set V^+ of vertices is traversed. We use locks for synchronization. Only the vertices in V^+ are locked, and their associated edges are not necessarily locked, which allows 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] V. Batagelj and M. Zaversnik, "An $O(m)$ algorithm for cores decomposition of networks," *CoRR*, vol. cs.DS/0310049, 2003. [Online]. Available: http://arxiv.org/abs/cs/0310049

[2] Y.-X. Kong, G.-Y. Shi, R.-J. Wu, and Y.-C. Zhang, "k-core: Theories and applications," *Physics Reports*, vol. 832, pp. 1–32, 2019.

[3] P. Wang, X. Deng, Y. Liu, L. Guo, J. Zhu, L. Fu, Y. Xie, W. Li, and J. Lai, "A knowledge discovery method for landslide monitoring based on k-core decomposition and the louvain algorithm," *ISPRS International Journal of Geo-Information*, vol. 11, no. 4, p. 217, 2022.

[4] R. Dorantes-Gilardi, D. García-Cortés, E. Hernández-Lemus, and J. Espinal-Enríquez, "k-Core genes underpin structural features of breast cancer," *Scientific Reports*, vol. 11, no. 1, pp. 1–17, 2021.

[5] P. Gao, J. Huang, and Y. Xu, "A k-core decomposition-based opinion leaders identifying method and clustering-based consensus model for large-scale group decision making," *Computers & Industrial Engineering*, vol. 150, p. 106842, 2020.

[6] K. Burleson-Lesser, F. Morone, M. S. Tomassone, and H. A. Makse, "k-core robustness in ecological and financial networks," *Scientific reports*, vol. 10, no. 1, pp. 1–14, 2020.

[7] F. D. Malliaros, C. Giatsidis, A. N. Papadopoulos, and M. Vazirgiannis, "The core decomposition of networks: Theory, algorithms and applications," *The VLDB Journal*, vol. 29, no. 1, pp. 61–92, 2020. [Online]. Available: https://doi.org/10.1007/s00778-019-00587-4

[8] J. Cheng, Y. Ke, S. Chu, and M. T. Özsu, "Efficient core decomposition in massive networks," in *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 2011, pp. 51–62.

[9] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo, "k-core decomposition of large networks on a single pc," *Proceedings of the VLDB Endowment*, vol. 9, no. 1, pp. 13–23, 2015.

[10] A. Montresor, F. De Pellegrini, and D. Miorandi, "Distributed k-core decomposition," *IEEE Transactions on parallel and distributed systems*, vol. 24, no. 2, pp. 288–300, 2012.

[11] D. Wen, L. Qin, Y. Zhang, X. Lin, and J. X. Yu, "I/o efficient core graph decomposition at web scale," in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 2016, pp. 133–144.

[12] D. Miorandi and F. De Pellegrini, "K-shell decomposition for dynamic complex networks," in *8th International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks*. IEEE, 2010, pp. 488–496.

[13] S. Pei, L. Muchnik, J. S. Andrade Jr, Z. Zheng, and H. A. Makse, "Searching for superspreaders of information in real-world social media," *Scientific reports*, vol. 4, p. 5547, 2014.

[14] K. Gabert, A. Pinar, and Ü. V. Çatalyürek, "Shared-memory scalable k-core maintenance on dynamic graphs and hypergraphs," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2021, pp. 998–1007.

[15] F. Zhang, B. Liu, and Q. Fang, "Core decomposition, maintenance and applications," in *Complexity and Approximation*. Springer, 2020, pp. 205–218.

[16] B. Guo and E. Sekerinski, "Simplified algorithms for order-based core maintenance," *arXiv preprint arXiv:2201.07103*, 2022.

[17] Y. Zhang, J. X. Yu, Y. Zhang, and L. Qin, "A fast order-based approach for core maintenance," in *Proceedings - International Conference on Data Engineering*, 2017, pp. 337–348.

[18] A. E. Sarıyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek, "Incremental k-core decomposition: algorithms and evaluation," *The VLDB Journal*, vol. 25, no. 3, pp. 425–447, 2016.

[19] H. Wu, J. Cheng, Y. Lu, Y. Ke, Y. Huang, D. Yan, and H. Wu, "Core decomposition in large temporal graphs," in *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 2015, pp. 649–658.

[20] A. E. Saríyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek, "Streaming algorithms for k-core decomposition," *Proceedings of the VLDB Endowment*, vol. 6, no. 6, pp. 433–444, 2013.

[21] R.-H. Li, J. X. Yu, and R. Mao, "Efficient core maintenance in large dynamic graphs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 10, pp. 2453–2465, 2013.

[22] Q.-S. Hua, Y. Shi, D. Yu, H. Jin, J. Yu, Z. Cai, X. Cheng, and H. Chen, "Faster parallel core maintenance algorithms in dynamic graphs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 6, pp. 1287–1300, 2019.

[23] H. Jin, N. Wang, D. Yu, Q. S. Hua, X. Shi, and X. Xie, "Core Maintenance in Dynamic Graphs: A Parallel Approach Based on Matching," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 11, pp. 2416–2428, nov 2018.

[24] N. Wang, D. Yu, H. Jin, C. Qian, X. Xie, and Q.-S. Hua, "Parallel algorithm for core maintenance in dynamic graphs," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 2366–2371.

[25] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.

[26] B. Guo and E. Sekerinski, "New parallel order maintenance data structure," *arXiv preprint arXiv:2208.07800*, 2022.
[27] J. JéJé, *An introduction to parallel algorithms*. Reading, MA: Addison-Wesley, 1992.
[28] N. S. Dasari, R. Desh, and M. Zubair, "Park: An efficient algorithm for k-core decomposition on multicore processors," in *2014 IEEE International Conference on Big Data (Big Data)*. IEEE, 2014, pp. 9–16.
[29] H. Kabir and K. Madduri, "Parallel k-core decomposition on multicore platforms," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2017, pp. 1482–1491.
[30] B. Sun, T.-H. H. Chan, and M. Sozio, "Fully dynamic approximate k-core decomposition in hypergraphs," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 14, no. 4, pp. 1–21, 2020.
[31] M. Yu, D. Wen, L. Qin, Y. Zhang, W. Zhang, and X. Lin, "On querying historical k-cores," *Proceedings of the VLDB Endowment*, vol. 14, no. 11, pp. 2033–2045, 2021.
[32] Z. Lin, F. Zhang, X. Lin, W. Zhang, and Z. Tian, "Hierarchical core maintenance on large dynamic graphs," *Proceedings of the VLDB Endowment*, vol. 14, no. 5, pp. 757–770, 2021.
[33] T. Weng, X. Zhou, K. Li, P. Peng, and K. Li, "Efficient distributed approaches to core maintenance on large dynamic graphs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 1, pp. 129–143, 2021.
[34] Q. C. Liu, J. Shi, S. Yu, L. Dhulipala, and J. Shun, "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.
[35] W. Zhou, H. Huang, Q.-S. Hua, D. Yu, H. Jin, and X. Fu, "Core decomposition and maintenance in weighted graph," *World Wide Web*, vol. 24, no. 2, pp. 541–561, 2021.
[36] B. Liu and F. Zhang, "Incremental algorithms of the core maintenance problem on edge-weighted graphs," *IEEE Access*, vol. PP, pp. 1–1, 04 2020.
[37] P. Dietz and D. Sleator, "Two algorithms for maintaining order in a list," in *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, 1987, pp. 365–372.
[38] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito, "Two simplified algorithms for maintaining order in a list," in *European Symposium on Algorithms*. Springer, 2002, pp. 152–164.
[39] R. Utterback, K. Agrawal, J. T. Fineman, and I.-T. A. Lee, "Provably good and practically efficient parallel race detection for fork-join programs," in *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, 2016, pp. 83–94.
[40] T. G. Mattson, Y. He, and A. E. Koniges, "The openmp common core," 2019.
[41] M. Herlihy, N. Shavit, V. Luchangco, and M. Spear, *The art of multiprocessor programming*. Newnes, 2020.