Revisiting Core Maintenance for Dynamic Hypergraphs

Qiang-Sheng Hua, *Member, IEEE*, Xiaohui Zhang, Hai Jin, *Fellow, IEEE*, and Hong Huang, *Member, IEEE*

***Abstract*—Core maintenance for dynamic hypergraphs has been receiving an increasing attention. However, existing works mainly focus on the insertion/deletion of hyperedges. This article revisits the problem from the view of vertices change. We study core maintenance when the vertices are inserted/deleted into/from specific hyperedges in the hypergraph, which is a challenging task since the deletion of the vertex may increase the core numbers and the insertion of the vertex may decrease the core numbers. We discuss in detail the possible changes of core numbers in different situations. For the insertion/deletion of vertices contained by a single hyperedge, we design sequential algorithms to discover the vertices whose core numbers have changed. Compared with static recomputation (Leng et al. 2013) and LYCLC (Luo et al. 2021) algorithms, our sequential algorithms can accelerate more than 1,000× and 12× at most in the processing time, respectively. For the insertion/deletion of vertices contained by different hyperedges, we find that core numbers of all vertices change 1 at most if these hyperedges form a matching. We design parallel algorithms that divide a matching into different sets based on their core numbers and allot a thread to each set. Experiments show that our parallel algorithms have good stability, scalability, and parallelism. Compared with the parallel static algorithm (Gabert et al. 2021) and the parallel dynamic algorithm GPC (Gabert et al. 2021), our parallel algorithms with 32 threads can accelerate 33× and 22× at most in the processing time, respectively.**

***Index Terms*—Core maintenance, dynamic hypergraphs, parallel algorithm.**

I. Introduction

MINING cohesive subgraphs in graphs is a critical task in graph analysis. There are a variety of cohesive subgraphs, e.g., k-core [3], k-peak [7], k-truss [14], [24], and (r, s)-nucleus [18]. Among them, k-core has attracted extensive attention since it can be calculated in linear time. For an unweighted and undirected graph, a k-core refers to the maximal subgraph where degrees of all vertices are not less than k. The concept closely related to k-core is core number. For a vertex u, its core number is k if it is in a k-core where k is the largest. k-core and core number are widely applied in many fields, including network modeling and analysis [8], anomaly detection [20], detection of influential spreaders [1], network visualization [27] and text analytics [23].

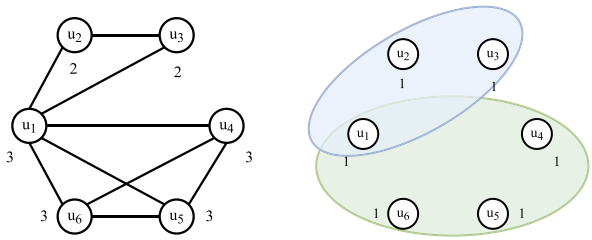


Fig. 1. u_1, u_2, u_3 are coauthors of one paper and u_1, u_4, u_5, u_6 coauthor another paper. The number next to the vertex represents the core number.

Unfortunately, the classic k-core model fails to depict polyadic relationships. More than two entities may participate in the same connection in many real-world problems. For example, the paper has several coauthors, the email has multiple recipients, and the club has numerous members. To better model polyadic relationships, researchers put forward the concept of the hypergraph. An unweighted and undirected hypergraph is composed of vertices and hyperedges, where each hyperedge can contain any number of vertices. We emphasize that vertices are *contained* rather than *connected* by hyperedges to distinguish hypergraphs from ordinary graphs. For a vertex u in the hypergraph, its degree is defined as the number of hyperedges containing the vertex u.

The concepts of k-core and core number can be easily extended to the hypergraph. Fig. 1(a) shows an ordinary graph and Fig. 1(b) is the hypergraph corresponding to it. u_1, u_2, u_3 are coauthors of one paper and u_1, u_4, u_5, u_6 coauthor another paper. What we model as hyperedges form 3-clique and 4-clique in the traditional model, respectively [22]. The core numbers of all vertices are two or three in Fig. 1(a), while the core numbers of all vertices are one in Fig. 1(b). In the ordinary graph model, the authors who have written a paper with many coauthors earn great impact, which obviously does not accord with the actual situation. Thus hypergraph k-core model can depict influential vertices better than ordinary graphs in multi-entity networks.

In the real world, neither ordinary graphs nor hypergraphs are invariant. In social networks, creating a new account means

Manuscript received 19 April 2022; revised 1 December 2022; accepted 1 January 2023. Date of publication 13 January 2023; date of current version 31 January 2023. This work was supported in part by National Natural Science Foundation of China under Grants 61972447 and 61832006. Recommended for acceptance by J. Zola. *(Corresponding author: Qiang-Sheng Hua.)*

The authors are with the National Engineering Research Center–Big Data Technology and System Lab, Key Laboratory of Services Computing Technology and System, Key Laboratory of Cluster and Grid Computing, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: qshua@hust.edu.cn; husterzxh@foxmail.com; hjin@hust.edu.cn; honghuang@hust.edu.cn).

This article has supplementary downloadable material available at https://doi.org/10.1109/TPDS.2023.3236669, provided by the authors.

Digital Object Identifier 10.1109/TPDS.2023.3236669

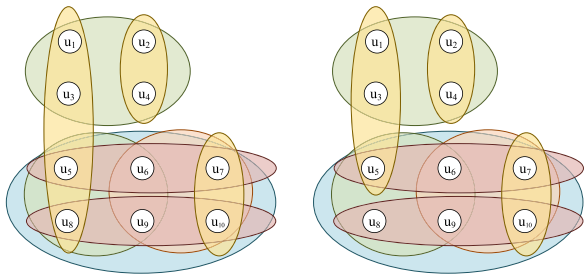
the formation of a new vertex, and following and unfollowing between users correspond to the appearance and disappearance of edges. Core numbers of vertices change with the change of the graphs/hypergraphs. Updating core numbers after changes is usually called core maintenance. The natural method of core maintenance is recomputing core numbers of all vertices in the updated graphs/hypergraphs, which needs a large time overhead. In fact, if the changed region is very small, only a few vertices need to have their core numbers updated.

Researchers usually only consider the insertion/deletion of edges when studying core maintenance in dynamic graphs because the insertion/deletion of vertices can be regarded as the insertion/deletion of all edges connected to them. Following this trend, some recent works [13], [22] on core maintenance in dynamic hypergraphs also focus on computing new core numbers after the insertion/deletion of hyperedges. However, when we revisit the problem, we can find another dynamic change in the hypergraph: the insertion/deletion of vertices in particular hyperedges, which may not cause the appearance and disappearance of the hyperedge since a hyperedge can contain any number of vertices. This change is meaningful. For example, Alice, a member of group A, group B, and group C, has now quit group C, corresponding to the hypergraph, which means that vertex Alice has been removed from the corresponding hyperedge of group C. But the hyperedge of group C may still exist since there are other members. In addition, group A and group B remain unchanged.

Core maintenance is a challenging task after vertices change in the hyperedges. Unlike the one-to-one correspondence between the insertion/deletion of hyperedges and the increase/decrease of core numbers, there is no significant relationship between the insertion/deletion of vertices in hyperedges and the increase/decrease of core numbers. In other words, even deleting a vertex in a hyperedge may increase the core numbers of other vertices. The trivial method is to rerun static core decomposition from scratch, which is not applicable since the cost of recalculation is expensive.

We note that the insertion/deletion of vertices in hyperedges is closely related to the insertion/deletion of hyperedges. The insertion of a hyperedge can be regarded as inserting vertices into a hyperedge that does not contain any vertex. The deletion of a hyperedge means that all vertices are deleted from the hyperedge. On the other hand, the insertion/deletion of vertices in hyperedges can also be regarded as deleting hyperedges and then inserting new hyperedges. Therefore, we can use the core maintenance algorithm for the insertion/deletion of hyperedges to deal with the insertion/deletion of vertices in hyperedges. However, this method also has a lot of redundant calculations.

Fig. 2 shows an example, and Table I lists all hyperedges in Fig. 2. The core numbers of u_1–u_4 are 2, and the core numbers of u_5–u_{10} are 3. To update core numbers after u_8 is deleted from e_1, we utilize the algorithms of hyperedge insertion/deletion to process the deletion of e_1 first and then the insertion of e_1' where $e_1' = \{u_1, u_3, u_5\}$. After e_1 is deleted, the core numbers of u_1–u_4 will be decreased by 1. After e_1' is inserted, the core numbers of u_1–u_4 will be restored to 2. In fact, if u_8 is deleted from e_1, the core numbers of all vertices will not increase or



(a) Original Hypergraph (b) Updated Hypergraph

Fig. 2. The core numbers of u_1–u_4 are 2 and the core numbers of u_5–u_{10} are 3 in the original hypergraph. If u_8 is deleted from e_1, all core numbers remain unchanged. But if the algorithms for the insertion/deletion of hyperedges perform, core numbers of u_1–u_4 decrease by 1 after e_1 is deleted and their core numbers restore to 2 after e_1' where $e_1' = \{u_1, u_3, u_5\}$ is inserted.

TABLE I
ALL HYPEREDGES IN FIG. 2(A)

hyperedge	vertices	hyperedge	vertices
e_1	u_1, u_3, u_5, u_8	e_2	u_1, u_2, u_3, u_4
e_3	u_2, u_4	e_4	u_5, u_6, u_8, u_9
e_5	u_6, u_7, u_9, u_{10}	e_6	u_5, u_6, u_7
e_7	u_8, u_9, u_{10}	e_8	u_7, u_{10}
e_9	$u_5, u_6, u_7, u_8, u_9, u_{10}$		

decrease. Thus the algorithms of hyperedge insertion/deletion make redundant calculations. Another challenge is that the core numbers may increase or decrease regardless of how the vertices in the hyperedge change. For example, in Fig. 2, if u_1 and u_3 are deleted from e_1, the core value of u_1–u_4 will decrease, and the core value of u_5–u_{10} will increase.

Note that a recent work by Gabert et al. [29] also paid attention to the insertion/deletion of vertices in hypergraphs. The work maintains a temporary core number for each vertex and then iteratively updates them. The vertices whose temporary core numbers changed will be put into a set with their neighbors. The temporary core numbers of all vertices in the set would be recalculated in the next iteration until all of them converge to the proper core numbers. Their parallel maintenance algorithm (abbreviated as GPC) initialized the temporary core numbers based on their original core numbers and the updated hyperedges. However, the initialized core numbers might be far away from their proper core numbers, leading to limited improvement compared against the parallel static algorithm where the temporary core number of each vertex is initialized as its vertex degree. In contrast, the algorithms we proposed in this paper tackle the problem from a different way which can efficiently identify the vertices whose core numbers will not change. As a result, our method can avoid many unnecessary computations.

After the vertices change, our core maintenance needs to: (1) identify the affected vertices; (2) judge how many the core numbers of affected vertices change. For the change of vertices in a single hyperedge, we analyze the sufficient conditions for increase or decrease of core numbers and give algorithms to identify the affected vertices. Further, to deal with the vertex

changes in multiple hyperedges, we prove that if these hyperedges form a matching, the core numbers of all vertices will only change by 1. In order to quickly identify the affected vertices, we use multithreading to accelerate the process. Previous works on core maintenance in ordinary graphs used matching [10], superior edge set [25] and joint edge set [9]. Superior edge set and joint edge set are not applicable to hypergraphs since their insertion/deletion cannot ensure that the core numbers change by 1 at most. It is not trivial to use matching in hypergraph core maintenance since there are vertices with increasing and decreasing core numbers at the same time. Our contributions are summarized as follows:

- For the insertion/deletion of vertices in the single hyperedge, we analyze the different situations and propose sequential algorithms to perform core maintenance.
- For the insertion/deletion of vertices in multiple hyperedges, we prove that if the hyperedges containing inserted/deleted vertices construct a matching, all vertices change their core numbers by 1 at most. We design parallel algorithms to solve this problem.
- Extensive experiments have presented that our sequential algorithms and parallel algorithms are are superior to the existing algorithms.

The rest of this paper is organized as follows. We give some formal definitions and theoretical basis in Sections II and III, respectively. The detailed implementation of algorithms is introduced in Section IV. The experimental setup and result analysis can be found in Section V. The related work is made in Section VI. Finally, we summarize the paper in Section VII.

II. Preliminaries

An unweighted and undirected hypergraph $H = (V, E)$ is a generalization of a graph, where V or $V(H)$ means a vertex set and E or $E(H)$ means a hyperedge set. In contrast to an ordinary edge which connects exactly two vertices, a hyperedge contains any number of vertices. We denote a hyperedge in E as e, or e_i for a specific one. For a vertex $u \in V(H)$, we denote a hyperedge that contains u as $e_H(u) \in E(H)$ and denote the set of all hyperedges that contain u as $E_H(u) \subseteq E(H)$. The degree of u, denoted as $d_H(u)$, is the number of hyperedges that u belongs to, i.e., $|E_H(u)|$. If there is no ambiguity, we omit the subscript for brevity. For example, we use $E(u)$ instead of $E_H(u)$. We denote a sub-hypergraph of H as S where $V(S) \subseteq V(H)$ and $E(S) \subseteq E(H)$. Some definitions are formally given to explain related concepts in the following.

Definition 1 (.k-Core). A k-core in an unweighted and undirected hypergraph H is defined as a maximal sub-hypergraph S in which the minimum degree of all vertices is k, i.e., $\forall u \in V(S), d_S(u) \geq k$.

Definition 2 (Core Number of a Vertex). The core number of a given vertex u in the hypergraph H is k if u is in a k-core where k is the largest. It is denoted as $vCore_H(u)$ or $vCore(u)$.

Definition 3 (Core Number of a Hyperedge). The core number of a given hyperedge e in the hypergraph H is k if the core number of each vertex that belongs to e is not less than k. It is denoted as $eCore_H(e)$ or $eCore(e)$. It satisfies that $eCore(e) = \min\{vCore(u) \mid u \in e\}$. If not particularly emphasized, the core number in the following refers to the core number of a vertex (not a hyperedge).

According to Definitions 2 and 3, we have the following equation to reveal the connection between the two.

$$vCore(u) = arg\max_{c \geq 0}\{|\{e \in E(u) | eCore(e) \geq c\}| \geq c\} \tag{1}$$

where c is a non-negative integer.

Eq. (1) implies that if the core number of a vertex is k, the vertex is contained by at least k hyperedges whose core numbers are equal to or greater than k.

Definition 4 (Pre-Core Number of a Hyperedge). Given a hyperedge e in the original hypergraph H and an updated hyperedge e' in the updated hypergraph H', we denote the pre-core number of e' as $preCore_{H'}(e')$ or $preCore(e')$. It satisfies that $preCore(e') = \min\{vCore_H(u) \mid u \in e'\}$. It is obvious that $eCore(e) \leq preCore(e')$ if some vertices are deleted from e and $eCore(e) \geq preCore(e')$ if some vertices are inserted into e. For example, after we delete u_1 and u_3 from e_1 in Fig. 2(a), $preCore(e'_1)$ is 3 where $e'_1 = \{u_5, u_8\}$.

It is necessary to point out that, throughout the paper, *we distinguish between before and after the change by adding prime in the upper right corner of the symbol*. For example, e and e' mean the original hyperedge and updated hyperedge, respectively.

Definition 5 (Core Maintenance). Core maintenance in dynamic hypergraphs is to update the core number for each vertex $u \in V(H')$ when the hypergraph $H(V, E)$ is updated to $H'(V, E')$. Note that the core number of each hyperedge can be computed from the vertices' core numbers easily.

We list some essential notations and their descriptions in Table II.

TABLE II
Notations and Their Descriptions

Notations	Descriptions
$V(H)$	the vertex set of hypergraph H
$E(H)$	the hyperedge set of hypergraph H
$vCore(u)$	the core number of vertex u (cf. Definition 2)
$eCore(e)$	the core number of hyperedge e (cf. Definition 3)
$preCore(e)$	the pre-core number of e (cf. Definition 4)
V_D, V_{Di}	the deletion vertex set in a hyperedge e, e_i
V_I, V_{Ii}	the insertion vertex set in a hyperedge e, e_i
e_{min}	the vertex set with the minmum core number in e
$e_{i_{min}}$	the vertex set with the minmum core number in e_i
$V_{D_{min}}$	the vertex set with the minimum core number in V_D
$V_{Di_{min}}$	the vertex set with the minimum core number in V_{Di}
$V_{I_{min}}$	the vertex set with the minimum core number in V_I
$V_{Ii_{min}}$	the vertex set with the minimum core number in V_{Ii}
$AD(u)$	the auxiliary degree of vertex u (cf. Definition 6)
E_M	a matching in the hyperedge set E (cf. Definition 9)

III. Theoretical Basis

When the hypergraph changes, the core numbers of many vertices remain unchanged, so it is unwise to start static calculation from scratch. In order to avoid a large number of redundant calculations, core maintenance needs to solve two

key problems: identifying potentially affected vertices and accurately judging the core number change of affected vertices. In this section, we give the theoretical basis for solving these two problems.

A. Core Number Change

Prior work has shown that core numbers of all vertices increase (*resp.* decrease) by 1 at most after a single hyperedge is inserted into (*resp.* deleted from) the original hypergraph. Since the insertion/deletion of vertices in a single hyperedge can be regarded as inserting a new hyperedge after the old hyperedge is deleted, the changed value of core numbers is at most 1. However, the insertion/deletion is not a simple one-to-one correspondence with the increase/decrease of core numbers. In other words, the deletion of vertices in a hyperedge may lead to a larger core number, and the insertion may lead to a smaller core number. According to the above analysis, an obvious solution is to utilize the ready-made core maintenance algorithms for the insertion/deletion of hyperedges. But the solution will cause a lot of redundant calculations. Therefore, solving the nontrival problem requires more efforts.

Through further observation, we find that whether the core numbers of vertices increase or decrease is closely related to the core numbers of inserted/deleted vertices and the pre-core numbers of the target hyperedges. When the deleted vertices meet certain conditions, the core numbers of all vertices will only decrease and remain unchanged. Inserting vertices has a similar conclusion.

Lemma 1. Given a hyperedge $e \in E(H)$ and a set of vetices $V_D \subseteq e$, let $e' = e \setminus V_D$ be the updated hyperedge. After V_D are deleted from e,

(i) if $eCore(e) = preCore(e')$, then core numbers of all vertices decrease by 1 or remain unchanged.

(ii) if $eCore(e) < preCore(e')$, then core numbers of all vertices will be changed by at most 1.

The proof of Lemma 1 can be found in Appendix A.1, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2023.3236669.

Lemma 2. Given a hyperedge $e \in E(H)$ and a set of vetices $V_I \subseteq V$ that $\forall u \in V_I$, $u \notin e$, let $e' = e \cup V_I$ be the updated hyperedge. After V_I are inserted into e,

(i) if $eCore(e) = preCore(e')$, then core numbers of all vertices increase by 1 or remain unchanged.

(ii) if $eCore(e) > preCore(e')$, then core numbers of all vertices will be changed by at most 1.

The proof of Lemma 2 is similar to that of Lemma 1 and will not be repeated here.

For example, after u_1 and u_3 are deleted from e_1 in Fig. 2(a), the core numbers of u_1–u_4 decrease by 1 and the core numbers of u_5–u_{10} increase by 1. The $eCore(e_1)$ is 2 and the $preCore(e'_1)$ is 3 where $e'_1 = \{u_5, u_8\}$, which meets case (ii) of Lemma 1. After core numbers of all vertices are updated, we insert u_1 and u_3 back into e'_1, which makes e'_1 restore as e_1. We have $eCore(e'_1) = 4$ and $preCore(e_1) = 1$. Case (ii) of Lemma 2 is satisfied so that the core numbers of some vertices increase and the core number of some vertices decrease.

Lemmas 1 and 2 answer the question of core number change in the case of vertices insertion/deletion in a single hyperedge.

B. Affected Vertices

Sariyüce et al. [17] provided the algorithms to identify the affected vertices when a single edge is inserted/deleted in an ordinary graph. Inspired by the algorithms, we design a scheme to identify the affected vertices when the vertices in the hyperedge are inserted/deleted. We first give several helpful definitions and then find the affected vertices through Lemmas 3 and 4.

Definition 6 (Auxiliary Degree). Given a vertex $u \in V(H)$, the auxiliary degree of u is the number of hyperedges containing u whose core numbers are equal to or greater than $vCore(u)$, which is denoted as $AD(u)$, i.e., $AD(u) = |\{e | e \in E_H(u) \wedge eCore(e) \geq vCore(u)\}|$. It is obvious that the auxiliary degree of a vertex cannot be less than its core number based on (1). We take u_5 with core number 3 in Fig. 2(a) as an example. u_5 is contained by e_1, e_4, e_6, e_9 and their core numbers are 2, 3, 3, 3, respectively. Thus the auxiliary degree of u_5 is 3.

The auxiliary degrees of all vertices are easy to be calculated if we know the core numbers of all vertices.

Definition 7 (Joint Common Vertices Set of a Path). The common vertices set between two hyperedges e_1 and e_2 refer to the intersection of e_1 and e_2. A path in a hypergraph is a sequence of hyperedges such that each consecutive hyperedge pair shares at least one vertex. Then the common vertices sets of the consecutive hyperedge pairs constitute the joint common vertices set of a path. For example, there exists a path (e_1, e_6, e_8) in Fig. 2(a) and the joint common vertices set is $\{\{u_5\}, \{u_7\}\}$.

Definition 8 (Reachable Sub-hypergraph of a Vertex). Given a vertex $u \in V(H)$ with core number k, a reachable sub-hypergraph of u is a sub-hypergraph of H, which satisfies two conditions:

(i) core numbers of all hyperedges in the sub-hypergraph are k.

(ii) a vertex in the sub-hypergraph with core number k has a path to u where the minimum core number of *each* element in the joint common vertices set is k. Note that each element in the joint common vertices set is also a set consisting of common vertices between two hyperedges.

We take Fig. 2(a) as an example to explain the concept of reachable sub-hypergraph. The reachable sub-hypergraph of u_1 with core number 2 contains e_1, e_2 and e_3. Even if the core number of e_6 is 2, it is not on the reachable sub-hypergraph of u_1, since the core number of the common vertex u_5 between e_1 and e_6 is not 2. It should be pointed out that the reachable sub-hypergraph of a group of vertices refers to the union of reachable sub-hypergraphs of each vertex in the group. The reachable sub-hypergraph of vertex u can be determined by depth-first search (DFS).

Lemma 3. Given a hyperedge $e \in E(H)$ and a set of vetices $V_D \subseteq e$, let $e' = e \setminus V_D$ be the updated hyperedge. The vertices with the minimum core number of V_D and e' are denoted as $V_{D_{min}}$ and e'_{min}, respectively. After V_D are deleted from e,

(i) if the core number of a vertex decreases, it is contained by the reachable sub-hypergraph of $V_{D_{min}}$.

(ii) if the core number of a vertex increases, it is contained by the reachable sub-hypergraph of e'_{min}.

The proof of Lemma 3 can be found in Appendix A.2, available in the online supplemental material.

Lemma 4. Given a hyperedge $e \in E(H)$ and a set of vetices $V_I \subseteq V(H)$ where $\forall u \in V_I$, $u \notin e$. Let $e' = e \cup V_I$ be the updated hyperedge. The vertices with the minimum core number of V_I and e are denoted as $V_{I_{min}}$ and e_{min}, respectively. After V_I are inserted into e,

(i) if the core number of a vertex increases, it is contained by the reachable sub-hypergraph of $V_{I_{min}}$.

(ii) if the core number of a vertex decreases, it is contained by the reachable sub-hypergraph of e_{min}.

The proof of Lemma 4 is similar to that of Lemma 3 and will not be repeated here.

C. Parallel Batch Processing

We consider the insertion/deletion of vertices contained by different hyperedges. After inserting/deleting vertices into/from different hyperedges, it is difficult to judge how many the core numbers change. Fortunately, we find that inserted/deleted vertices can be divided into different batches such that the core numbers change by 1 at most after we insert or delete each batch.

Definition 9 (Matching). Given a set of hyperedges E_M, if $e_1 \cap e_2 = \emptyset$ where $\forall e_1 \in E_M, e_2 \in E_M$, then E_M is a matching. For example, e_1, e_3, and e_8 form a matching in Fig. 2(a).

Lemma 5. Given a matching E_M,

(i) core numbers of all vertices increase by 1 at most if E_M are inserted into the original hypergraph H.

(ii) core numbers of all vertices decrease by 1 at most if E_M are deleted from the original hypergraph H.

The proof of Lemma 5 can be found in Appendix A.3, available in the online supplemental material.

Lemma 6. Given a vertex set $\mathbb{V}_\mathbb{D} = \{\ldots, V_{Di}, \ldots\}$ and a hyperedge set $\mathbb{E}_\mathbb{D} = \{\ldots, e_i, \ldots\}$ where V_{Di} is the deletion vertex set in a hyperedge $e_i \in E(H)$ and $V_{Di} \subseteq e_i$. If the original hyperedge set $\mathbb{E}_\mathbb{D}$ is a matching, then core numbers of all vertices change by 1 at most after $\mathbb{V}_\mathbb{D}$ is deleted from $\mathbb{E}_\mathbb{D}$.

Lemma 7. Given a vertex set $\mathbb{V}_\mathbb{I} = \{\ldots, V_{Ii}, \ldots\}$ and a hyperedge set $\mathbb{E}_\mathbb{I} = \{\ldots, e_i, \ldots\}$ where V_{Ii} is the insertion vertex set in a hyperedge $e_i \in E(H)$ and $V_{Ii} \cap e_i = \emptyset$. If the updated hyperedge set $\mathbb{E}'_\mathbb{I}$ is a matching, then core numbers of all vertices change by 1 at most after $\mathbb{V}_\mathbb{I}$ is inserted into $\mathbb{E}_\mathbb{I}$.

It is obvious that Lemmas 6 and 7 can be derived from Lemma 5. Lemmas 6 and 7 provide a theoretical basis to judge how many the core numbers change in the case of inserting/deleting vertices into/from different hyperedges. Next, we focus on identifying the potentially affected vertices. Similar to the insertion/deletion of vertices in a single hyperedge, the potentially affected vertices are in the reachable sub-hypergraph.

Lemma 8. Given a vertex set $\mathbb{V}_\mathbb{D} = \{\ldots, V_{Di}, \ldots\}$ and a hyperedge set $\mathbb{E}_\mathbb{D} = \{\ldots, e_i, \ldots\}$ where V_{Di} is the deletion vertex set in a hyperedge $e_i \in E(H)$ and $V_{Di} \subseteq e_i$. Let $V_{min} = \{\cdots \cup V_{Di_{min}} \cup \cdots\}$ where $V_{Di_{min}}$ is the vertex set with the minimum core number in V_{Di} and $E_{min} = \{\cdots \cup e'_{i_{min}} \cup \cdots\}$ where $e'_{i_{min}}$ is the vertex set with the minimum core number in e'_i. If the original hyperedge set $\mathbb{E}_\mathbb{D}$ is a matching, after deleting $\mathbb{V}_\mathbb{D}$,

(i) if the core number of a vertex decreases, it is contained by the reachable sub-hypergraph of V_{min}.

(ii) if the core number of a vertex increases, it is contained by the reachable sub-hypergraph of E_{min}.

Lemma 9. Given a vertex set $\mathbb{V}_\mathbb{I} = \{\ldots, V_{Ii}, \ldots\}$ and a hyperedges set $\mathbb{E}_\mathbb{I} = \{\ldots, e_i, \ldots\}$ where V_{Ii} is the insertion vertex set in a hyperedge $e_i \in E(H)$ and $V_{Ii} \cap e_i = \emptyset$. Let $V_{min} = \{\cdots \cup V_{Ii_{min}} \cup \cdots\}$ where $V_{Ii_{min}}$ is the vertex set with the minimum core number in V_{Ii} and $E_{min} = \{\cdots \cup e_{i_{min}} \cup \cdots\}$ where $e_{i_{min}}$ is the vertex set with the minimum core number of e_i. If the updated hyperedge set $\mathbb{E}'_\mathbb{I}$ is a matching, after inserting $\mathbb{V}_\mathbb{I}$,

(i) if the core number of a vertex increases, it is contained by the reachable sub-hypergraph of V_{min}.

(ii) if the core number of a vertex decreases, it is contained by the reachable sub-hypergraph of E_{min}.

The proofs of Lemmas 8 and 9 are omitted since they are similar to Lemmas 3 and 4, respectively. The core numbers of hyperedges in the same reachable sub-hypergraph are the same, which ensures that reachable sub-hypergraphs with different core numbers will not overlap each other. The vertices in a batch (matching) are divided into different sets according to their core numbers, and the core numbers of vertices in the same set are the same. For parallel processing, we will assign a thread to each set which can avoid resource consumption and performance loss caused by locking and unlocking since different threads do not compete with each other.

Parallel batch processing brings a new issue. A vertex may be contained by the reachable sub-hypergraphs with both increased and decreased core numbers at the same time. For example, e_1 and e_8 form a matching in Fig. 2(a). We delete $\{u_1, u_3\}$ from e_1 and delete $\{u_7, u_{10}\}$ from e_8. All core numbers change by 1 at most based on Lemma 6. In fact, the core numbers of u_1–u_4 decrease 1 and those of other vertices don't change. The reachable sub-hypergraph of $\{u_5, u_8\}$ and that of $\{u_7, u_{10}\}$ overlap, but the core numbers of the former may decrease and those of the latter may increase. In response to this situation, the algorithms first identify the vertices with decreased core numbers, then identify the vertices with increased core numbers.

IV. Implementation

In Section III, we settle down the issue of which vertices are affected by the insertion/deletion and how their core numbers would change. In this section, we present the algorithms of core maintenance in dynamic hypergraphs based on the findings in Section III. We consider two scenarios, one is a sequential algorithm for the insertion/deletion of vertices in a single hyperedge, and the other is a parallel batch algorithm for the insertion/deletion of vertices in different hyperedges.

A. Insertion/Deletion of Vertices Contained by a Single Hyperedge

Algorithm 1 gives the basic process of core maintenance for the deletion of vertices contained by a single hyperedge, which

Algorithm 1: Vertdeletion.

Input:
$H = (V, E)$ is the original hypergraph.
V_D is the vertices to be deleted.
e is the hyperedge containing V_D.
The core numbers $vCore$ of all vertices before deletion.

1 $V_{D_{min}} \leftarrow$ the vertex set with the minimum core number in V_D;
2 $e'_{min} \leftarrow$ the vertex set with the minimum core number in e';
3 $H' = (V, E')$ is the updated hypergraph;
4 **if** $eCore(e) = preCore(e')$ **then**
5 $\quad V_{dec} \leftarrow$ deleteFunc(H', $V_{D_{min}}$, $vCore$);
6 **else**
7 $\quad V_{dec} \leftarrow$ deleteFunc(H', $V_{D_{min}}$, $vCore$);
8 $\quad V_{inc} \leftarrow$ insertFunc(H', e'_{min}, $vCore$);
9 **for** $u \in V_{dec}$ **do**
10 $\quad vCore(u) \leftarrow vCore(u) - 1$;
11 **for** $u \in V_{inc}$ **do**
12 $\quad vCore(u) \leftarrow vCore(u) + 1$;

Algorithm 2: Vertinsertion.

Input:
$H = (V, E)$ is the original hypergraph.
V_I is the insertion vertices set.
e is the target hyperedge of V_I.
The core numbers $vCore$ of all vertices before insertion.

1 $V_{I_{min}} \leftarrow$ the vertex set with the minimum core number in V_I;
2 $e_{min} \leftarrow$ the vertex set with the minimum core number in e;
3 $H' = (V, E')$ is the updated hypergraph;
4 **if** $eCore(e) = preCore(e')$ **then**
5 $\quad V_{inc} \leftarrow$ insertFunc(H', $V_{I_{min}}$, $vCore$);
6 **else**
7 $\quad V_{inc} \leftarrow$ insertFunc(H', $V_{I_{min}}$, $vCore$);
8 $\quad V_{dec} \leftarrow$ deleteFunc(H', e_{min}, $vCore$);
9 **for** $u \in V_{dec}$ **do**
10 $\quad vCore(u) \leftarrow vCore(u) - 1$;
11 **for** $u \in V_{inc}$ **do**
12 $\quad vCore(u) \leftarrow vCore(u) + 1$;

first initializes the necessary variables and then discusses two circumstances according to Lemma 1: (1) when $eCore(e) = preCore(e')$, we only need to consider the decrease of core numbers; (2) otherwise, it is necessary to consider the increase of core numbers. In lines 4–8, Algorithm 1 calls Algorithms 3 and 4 to discover the vertices with decreased core numbers and the vertices with increased core numbers, respectively. The $Root$ parameters that are passed into Algorithms 3 and 4 are determined by Lemma 3. Finally in lines 9–12, the core numbers of the vertices in the V_{dec} and V_{inc} are updated. Algorithm 2 deals with core maintenance in the case of insertion, and the basic process is similar to Algorithm 1.

Algorithm 3 searches for vertices with decreased core numbers in the reachable sub-hypergraphs of $Root$. The local variable $vFlag$ indicates whether the vertex has been visited and $rFlag$ is used to track whether the core numbers of vertices decrease. In lines 6–28, for each vertex $u \in Root$ whose core number is k, we perform a DFS whose start vertex is u in the reachable sub-hypergraph of u. Each vertex maintains td, the number of hyperedges that contain the vertex and could contribute to the core number of the vertex. We initialize the td of each vertex as AD. For a vertex in the reachable sub-hypergraph, if its td is less than k, which indicates that its core number will decrease, we push it into the stack, and the td values of its neighbor vertices will decrease by 1. Finally, the vertices whose $vFlag$ and $rFlag$ are $true$ will be returned. Their core numbers decrease by 1.

Algorithm 4 searches for vertices with increased core numbers in the reachable sub-hypergraph. The function of the local variable $vFlag$ is the same as that of Algorithm 3. The other local variable $rFlag$ is used to mark the evicted vertices whose core numbers will not increase. Different from Algorithms 3, 4 has two DFS processes. In lines 12–19, the first DFS takes the vertex $u \in Root$ whose core number is k as the start vertex and seeks the vertices whose td values are greater than k in the reachable sub-hypergraph. Their core numbers may increase. When encountering a vertex v with $td[v] < k$, the td values of its neighbors decrease by 1. In lines 21–32, the second DFS is conducted in the reachable sub-hypergraph to spread the impact to other vertices. Finally, the vertices whose $vFlag$ are $true$ and $rFlag$ are $false$ are returned. Their core numbers increase by 1.

Theorem 10. Algorithms 1 and 2 can correctly update core numbers after the insertion/deletion of vertices contained by a single hyperedge.

Proof. According to Lemma 1, Algorithms 1 and 2 cover two possible circumstances and and correctly determine core numbers' changes. Then according to Lemma 3, Algorithms 3 and 4 update the reachable sub-hypergraph of *Root* correctly, ending in updating core numbers correctly.

Performance Analysis. To analyze the time complexity of our sequential algorithms (Algorithms 1 and 2), we need to introduce some notations.

For a hypergraph H, we assume that each hyperedge has f vertices at most and each vertex is contained by d_{max} hyperedges at most. Let $V_k(H)$ be the vertex set with core number k and $E_k(H)$ be the hyperedge set with core number k. Let $n_k = |V_k(H)|$ and $m_k = |E_k(H)|$. In addition, n represents the number of all vertices in the hypergraph.

Both Algorithms 1 and 2 call Algorithms 3 and 4. We need to analyze the time complexity of Algorithms 3 and 4. Algorithm 3 calculates the AD values of all vertices with core numbers k, whose time complexity is $O(d_{max} * n_k)$. Then performing DFS in the reachable sub-hypergraph takes $O(d_{max} * n_k + f * m_k)$.

Finally, the cost time of checking each vertex in V is $O(|n|)$. So the total time complexity is $O(d_{max} * n_k + f * m_k + n)$. The time complexity of Algorithm 4 is same as that of Algorithm 3. According to above results, the time complexity of Algorithms 1 and 2 is $O(d_{max} * n_k + f * m_k + n)$, which is almost same as that of LYCLC algorithms ($O(d_{max} * n_k + f * m_k)$). It should be pointed out that the above analysis is the worst case. Since our algorithms can accurately identify the case where all core numbers do not change at a small cost, the overhead is reduced. The experiment results show that our sequential algorithms are superior to LYCLC algorithms.

B. Insertion/Deletion of Vertices Contained by Different Hyperedges

In this subsection, we design parallel core maintenance algorithms for the insertion/deletion of vertices contained by different hyperedges. We take the deletion case as an example to illustrate the algorithm implementation in detail. The insertions case is similar to deletions, so they are not repeated to save space.

Algorithm 6 shows specific procedures, which consist of two parts. Initially, a preprocessing divides hyperedges containing deleted vertices into different maximal matchings in line 1. Note that, finding the maximal matchings can be done by a trivial greedy strategy. Next, the algorithm searches for the vertices whose core numbers may change for each matching.

We know that if the hyperedges containing deleted vertices construct a matching, then the core numbers of all vertices in the hypergraph change by 1 at most, as shown by Lemma 6. Given a vertex set $\mathbb{V}_{\mathbb{D}} = \{\ldots, V_{Di}, \ldots\}$ and a hyperedge set $\mathbb{E}_{\mathbb{D}} = \{\ldots, e_i, \ldots\}$ where V_{Di} is the deletion vertex set in a hyperedge $e_i \in E(H)$ and $V_{Di} \subseteq e_i$. We can find a matching E_M from $\mathbb{E}_{\mathbb{D}}$. $\mathcal{V}_{\mathcal{DM}}$ is a subset of $\mathbb{V}_{\mathbb{D}}$ where $V_{Di} \in \mathbb{V}_{\mathbb{D}}$ is the deletion vertex set in $e_i \in E_M$. $\mathcal{V}_{\mathcal{DM}_{min}}$ is the vertex set with the minimum core number of each set in $\mathcal{V}_{\mathcal{DM}}$. $E'_{M_{min}}$ is the vertex set with the minimum core number of each hyperedge in E'_M. According to Lemma 8, the vertices whose core numbers may decrease are contained by the reachable sub-hypergraph of $\mathcal{V}_{\mathcal{DM}_{min}}$ and the vertices that may increase their core numbers are contained by the reachable sub-hypergraph of $E'_{M_{min}}$. In order to process in parallel, Algorithm 6 divides $\mathcal{V}_{\mathcal{DM}_{min}}$ into different sets according to their core numbers in lines 7–10. Then it assigns a thread to each set, and calls Algorithm 3 to find the vertices with decreased core numbers. A similar process is conducted for $E'_{M_{min}}$ to find the vertices with increased core numbers in lines 13–16.

Theorem 11. Algorithm 6 can correctly update core numbers after the deletion of vertices contained by different hyperedges.

Proof. Algorithm 6 divides affected hyperedges into different maximal matchings to guarantee that core numbers of all vertices change by 1 at most, which is proven in Lemma 6. Then according to Lemma 8, the vertices whose core numbers may decrease or increase are separated into different vertex sets induced by reachable sub-hypergraphs. Similar to Theorem 10, Algorithms 3 and 4 are called to find out the correct core values.

Algorithm 3: Deletefunc.

Input:
$H' = (V, E')$ is the updated hypergraph.
Root is the root vertices set.
The core numbers *vCore* of all vertices before deletion.
Output:
V_{dec}, i.e., the vertices whose core numbers will decrease.

```
1  for each e in E' do
2      eFlag[e] ← false;
3  for each u in V do
4      td[u] ← 0; AD[u] ← 0;
5      vFlag[u] ← false; rFlag[u] ← false;
6  for each u in Root do
7      k ← vCore(u);
8      if vFlag[u] = false then
9          vFlag[u] ← true;
10         AD[u] ← ComputeAD(u);
11         td[u] ← AD[u] + td[u];
12     if td[u] < k and rFlag[u] = false then
13         rFlag[u] ← true;
14         stk is an empty stack;
15         stk.push(u);
16         while stk.empty() = false do
17             v ← stk.pop();
18             for each e ∈ E(v) with eCore(e) = k and
                eFlag[e] = false do
19                 eFlag[e] ← true;
20                 for each y ∈ e with vCore(y) = k do
21                     if vFlag[y] = false then
22                         vFlag[y] ← true;
23                         AD[y] ← ComputeAD(y);
24                         td[y] ← AD[y] + td[y];
25                     td[y] ← td[y] − 1;
26                     if rFlag[y] = false and td[y] < k
                        then
27                         stk.push(y);
28                         rFlag[y] ← true;
29 for each u in V do
30     if rFlag[u] = true and vFlag[u] = true then
31         V_dec ← V_dec ∪ {u};
32 return V_dec;
```

Performance Analysis. To analyze the time complexity of Algorithm 6, we continue to use the symbols used in the performance analysis of sequential algorithms. In addition, β represents the number of matchings in $\mathbb{E}_{\mathbb{D}}$. Let n_E and m_E be the number of vertices and hyperedges in $\mathbb{E}_{\mathbb{D}}$, respectively. We assume that each hyperedge has f^E vertices at most and each vertex is contained by d^E_{max} at most in $\mathbb{E}_{\mathbb{D}}$.

Algorithm 4: InsertFunc.

Input:
$H' = (V, E')$ is the updated hypergraph.
Root is the root vertices set.
The core numbers *vCore* of all vertices before insertion.
Output:
V_{inc}, i.e., the vertices whose core numbers will increase.

```
for each e in E' do
    eFlag[e] = false;
for each u in V do
    td[u] ← 0; AD[u] ← 0;
    vFlag[u] ← false; rFlag[u] ← false;
for each u with vFlag[u] = false in Root do
    k ← vCore(u); vFlag[u] ← true;
    AD[u] ← ComputeAD(u); td[u] ← AD[u] + td[u];
    stk1 is an empty stack; stk1.push(u);
    while stk1.empty() = false do
        v ← stk1.pop();
        if td[v] > k then
            for each e ∈ E(v) with eCore(e) = k do
                for each y ∈ e with vCore(y) = k do
                    if vFlag[y] = false then
                        AD[y] ← ComputeAD(y);
                        td[y] ← AD[y] + td[y];
                        vFlag[y] ← true;
                        stk1.push(y);
        else
            if rFlag[v] = false then
                stk2 is an empty stack;
                stk2.push(v); rFlag[v] ← true;
                while stk2.empty() = false do
                    y ← stk2.pop();
                    for each e ∈ E(y) with eCore(e) = k
                     and eFlag[e] = false do
                        eFlag[e] ← true;
                        for each z ∈ e with vCore(z) = k
                         do
                            td[z] ← td[z] − 1;
                            if rFlag[z] = false and
                             td[z] = k then
                                rFlag[z] ← true;
                                stk2.push(z);
for each u in V do
    if rFlag[u] = false and vFlag[u] = true then
        V_inc ← V_inc ∪ {u};
return V_inc;
```

We first analyze the time complexity of finding matchings. All hyperedges in $\mathbb{E}_{\mathbb{D}}$ are divided into d^E_{max} matchings at most. So the time complexity is $O(d^E_{max} * m_E * f^E)$, denoted by T_{pre}. We denote the time complexity of each iterator in Algorithm 6 as T_i such that $T_i = max\{d_{max} * n_k + f * m_k + n\}$ where k is a non-negative integer. The total time complexity is $O(T_{pre} + \sum_{i=1}^{\beta} T_i)$.

Algorithm 5: ComputeAD.

Input:
u is the vertex whose AD needs to be computed.
Output:
AD value of u.

```
AD ← 0;
for each e ∈ E(u) do
    if eCore(e) ≥ vCore(u) then
        AD ← AD + 1;
return AD;
```

Algorithm 6: ParallelVertDeletion.

Input:
$H = (V, E)$ is the original hypergraph.
$\mathbb{E}_{\mathbb{D}}$ is $\{\cdots, e_i, \cdots\}$ where $e_i \in E(H)$.
$\mathbb{V}_{\mathbb{D}}$ is $\{\cdots, V_{Di}, \cdots\}$ where V_{Di} is the deletion vertex set of e_i.
The core numbers *vCore* of all vertices before deletion.

```
E_Ms ← a set of matchings in E_D;
for each E_M in E_Ms do
    V_DM ← a subset of V_D where V_Di ∈ V_D is the
     deletion vertex set in e_i ∈ E_M;
    V_DM_min ← the vertex set with the minimum core
     number in each set in V_DM;
    H' ← the updated hypergraph where V_DM are
     deleted from E_M;
    V^k_DM_min ← the vertex set with core number k in
     V_DM_min;
    for V^k_DM_min with different k in parallel do
        V_k ← deleteFunc(H', V^k_DM_min, vCore);
        for u in V_k do
            vCore(u) ← vCore(u) − 1;
    E'_M_min ← the vertex set with the minimum core
     number of each hyperedge in E'_M;
    E'^k_M_min ← the vertex set with core number k in
     E'_M_min;
    for E'^k_M_min with different k in parallel do
        V_k ← insertFunc(H', E'^k_M_min, vCore);
        for u in V_k do
            vCore(u) ← vCore(u) + 1;
```

V. Evaluation

We design thorough experiments on 12 real-world hypergraphs to evaluate our proposed algorithms. First, we test the stability and scalability of our algorithms. Next, the parallelism of the parallel algorithms is shown. Finally, we compare our algorithms with existing algorithms, including the sequential static core decomposition [12], LYCLC algorithms [13], the parallel static algorithm and the parallel dynamic GPC algorithm [29]. The first utilizes the idea of the peeling algorithm [19]. LYCLC extends the core maintenance algorithms in ordinary graphs [17]

to hypergraphs. The latter two parallel algorithms are based on the local h-index algorithm [31]. We utilize C++ 11 to implement all algorithms. All source codes are compiled by g++ 9.3.0 with -O3 level optimization. The evaluations are performed on a Linux machine with 56 Intel Xeon E5-2680@2.40 GHz CPUs and 250 GB main memory.

The algorithms read two files from the disk, the original hypergraph, and the changed hyperedges. Each row representing a hyperedge in the original hypergraph has several unsigned integers representing vertices that are contained by the hyperedge. The first unsigned integer in each row of the updated hypergraph represents a hyperedge. The following unsigned integers represent the vertices, indicating that these vertices are inserted/deleted in the hyperedge. We first delete these vertices from the specific hyperedges to test deletion algorithms, and then insert them back into original hyperedges to test insertion algorithms. We use two 2-D $vector$ data structure to store the hypergraph. All vertices that make up a hyperedge are stored in the first $vector$. The elements in the second $vector$ represent all hyperedges containing the vertex. We design a thread pool to implement the parallelization of the algorithm. The thread pool determines the number of threads through the parameters from the command line. There are a thread queue and a task queue in the thread pool. One thread executes one task. When the thread finishes processing the current task, it will fetch a task from the task queue for execution.

For our sequential algorithms, the LYCLC algorithms, the sequential and parallel static core decomposition algorithms, and the parallel dynamic GPC algorithm, we record the sum of the time for updating hyperedges and the time for core decomposition or maintenance. For our parallel algorithms, in addition to the above time, the preprocessing time for finding matchings in updated hyperedges needs to be added. In the following experiments, we use *cost time* as an indicator to evaluate the efficiency of the algorithms. We use *time per hyperedge* when evaluating the influence of the size of the updated hyperedge set. In order to eliminate interference factors, we repeated all the experiments five times and took the average values as the final results. In the following figures, the units of the y-axis are milliseconds, abbreviated as ms. We compare the core numbers of all vertices updated by our algorithms with the correct core numbers calculated by the static algorithm [12] to verify the correctness of our algorithms.

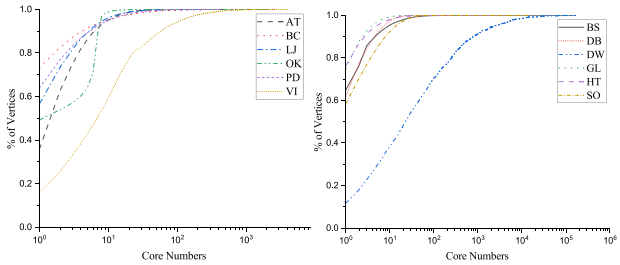
Datasets. Table III shows 12 real-world datasets, which can be accessed and downloaded from KONECT[1] and CORNELL.[2] These graphs are chosen from domains representing social networks, tagging networks, and authorship networks. AT, BC, LJ, OK, PD, and VI are static hypergraphs, and other hypergraphs are temporal hypergraphs where each hyperedge is associated with a time stamp. In the table, *accu.v* refers to the sum of the number of vertices contained by all hyperedges in the hypergraph, and *max k* refers to the maximum core number of all vertices in the hypergraph. Fig. 3 shows the distribution of core numbers in these hypergraphs. The core numbers are in a log axis. For most hypergraphs, the core numbers of more than half of the vertices are only 1.

[1][Online]. Available: http://konect.cc/networks/

[2][Online]. Available: https://www.cs.cornell.edu/~arb/data/

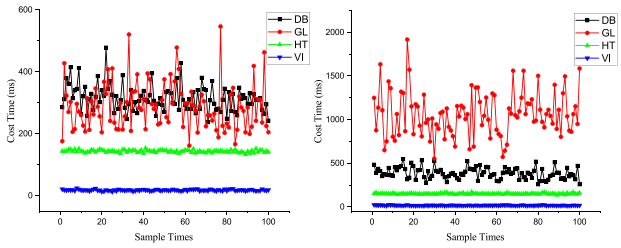
TABLE III
ATTRIBUTES OF DATASETS

Dataset	$\|V\|$	$\|E\|$	accu. v	max k
Actor2(AT)	304K	896K	3.78M	634
BookCrossing(BC)	105K	341K	1.15M	2,705
BibSonomy(BS)	205K	767K	2.56M	165,382
coauth-DBLP(DB)	1.92M	3.70M	10.3M	313
DAWN(DW)	2.56K	2.27M	3.60M	182,722
coauth-Geology(GL)	1.26M	1.59M	4.42M	564
coauth-History(HT)	1.01M	1.81M	2.37M	4,348
LiveJournalG(LJ)	3.20M	7.49M	112M	300
OrkutG(OK)	2.78M	8.73M	327M	2,100
Producers(PD)	48.8K	139K	207K	432
stack-overflow(SO)	2.68M	11.3M	25.6M	13,357
vi.sualize.us(VI)	17.1K	495K	2.30M	4,241



(a) Static Datasets (b) Temporal Datasets

Fig. 3. Core number distribution of real-world datasets.



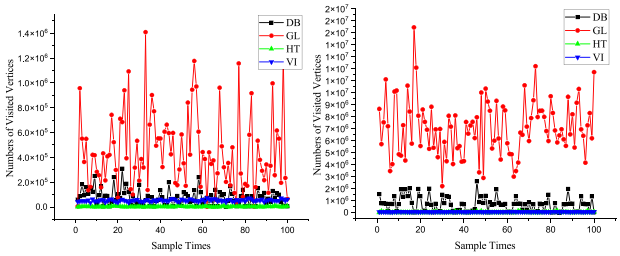
(a) Deletion (b) Insertion

Fig. 4. The stability of our sequential algorithms.

A. Stability Evaluation

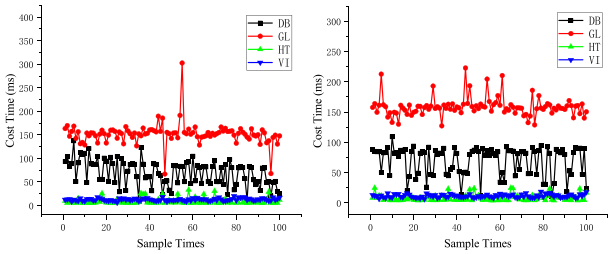
After the vertices in the hyperedges are inserted/deleted, we update the core numbers of the affected vertices. One issue is whether the efficiency of the sequential and parallel algorithms is stable if the updated hyperedge sets come from different regions of the hypergraph. For sequential algorithms, we randomly sample 100 hyperedges and select several vertices in each hyperedge as updated vertices. We first delete these vertices from the hyperedges and then insert them back into the hyperedges. After each change occurs, the corresponding algorithm is called to maintain the core numbers. We accumulate the total time to process 100 selected hyperedges. Finally, we repeat the above steps 100 times and get 100 sets of results.

Fig. 4(a) and (b) show the results of deletion and insertion, respectively. The x-axis represents the number of samples and



(a) Deletion (b) Insertion

Fig. 5. The number of visited vertices for our sequential algorithms.



(a) Deletion (b) Insertion

Fig. 6. The stability of our parallel algorithms.

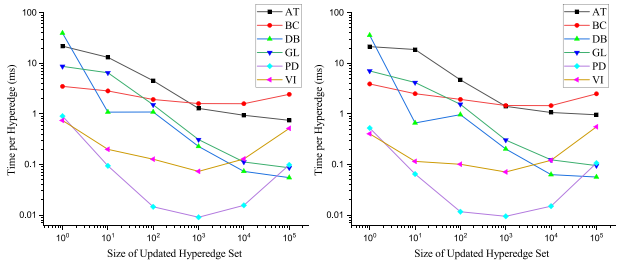
TABLE IV
STATISTICAL MEASURES OF COST TIME IN STABILITY EVALUATION

Sample	*Mean*	*St.Dev.*	Sample	*Mean*	*St.Dev.*
for our sequential algorithms					
DB del.	314.98	42.563	DB ins.	387.05	66.138
GL del.	286.25	76.370	GL ins.	1063.2	255.98
HT del.	143.36	3.9176	HT ins.	150.95	5.8697
VI del.	18.106	2.1668	VI ins.	17.753	2.5681
for our parallel algorithms					
DB del.	69.353	27.126	DB ins.	69.518	23.784
GL del.	151.82	22.626	GL ins.	158.42	15.506
HT del.	8.9843	6.5423	HT ins.	8.2007	5.4528
VI del.	12.407	2.5859	VI ins.	11.417	2.5844

the y-axis is the cost time of core maintenance for the sample. Sequential algorithms show good stability on HT and VI. For DB and GL, we observe that their results fluctuate frequently. In fact, the cost time is positively correlated with the total number of visited vertices, which is shown in Fig. 5(a) and (b). If the number of visited vertices varies greatly, the cost time fluctuates frequently. The number of visited vertices for insertion and deletion is different for the same sample because the insertion algorithm performs two DFS processes and the deletion algorithm does it only once.

For our parallel algorithms, we set the number of threads to 8 and then evaluate the stability of the algorithms in the same way as the sequential algorithms. The results are shown in Fig. 6(a) and (b). Similarly, our parallel algorithms perform good stability on HT and VI, while the stability on DB and GL is poor. The reason is the same as the sequential algorithms.

The means and standard deviations (St. Dev.) of cost time on 100 samples for our sequential and parallel algorithms are showed in Table IV. Del. is short for deletion and ins. is short for insertion. The units are milliseconds (ms).



(a) Deletion (b) Insertion

Fig. 7. Influence of the size of updated hyperedge set.

B. Scalability Evaluation

In this subsection, we change the size of the updated hyperedge set and graphs to measure the scalability of parallel algorithms. We change the size of the updated hyperedge set, fix the number of threads to 8, and record how long it takes to maintain the core numbers. The results are shown in Fig. 7(a) and (b). The x-axis and the y-axis represent the size of the updated hyperedge set and the ratio of the core maintenance time to the size of the updated set, respectively. Both the x-axis and the y-axis are logarithmic coordinates. For static hypergraphs, updated hyperedge sets are sampled randomly, and for temporal hypergraphs, we select latest hyperedges.

As can be seen from Fig. 7, with the exponential increase of size of updated hyperedge set, the average cost time per hyperedge shows a downward trend. This is because as the number of updated hyperedges increases, the algorithms can process more hyperedges in one iteration, thus reducing repeated access. In addition, the core numbers may be distributed evenly if the batch contains more hyperedges, which makes better use of multi-threading acceleration. For hypergraphs (PD, VI) with small vertices and hyperedges, when updated hyperedge set contains a large of hyperedges, average time for core maintenance increases. It is because that the changed region accounts for a large proportion of the whole hypergraph so that algorithms perform a DFS on the entire hypergraph to find affected vertices almost.

To evaluate the influence of the size of hypergraph, we sample hyperedges at rates from 10% to 100% to rebuild new hypergraph. For each new hypergraph, we further sample 1,000 (if the number of hyperedges in the hypergraph is less than one million) or 10,000 hyperedges for testing (if the number of hyperedges in the hypergraph is not less than one million). The results are shown in Fig. 8(a) and (b). The x-axis is the proportion of the sampled hypergraph to the initial hypergraph and the y-axis represents the cost time for core maintenance. Overall, the cost time increase with the increase of the sizes of hypergraphs since potentially affected vertices increase.

C. Parallelism Evaluation

We select four graphs AT, DB, SO, and VI to conduct the experiment of parallelism. For static graphs, we randomly sample 10,000 hyperedges. For temporal graphs, we directly select the latest 10,000 hyperedges (with maximum timestamps). Several

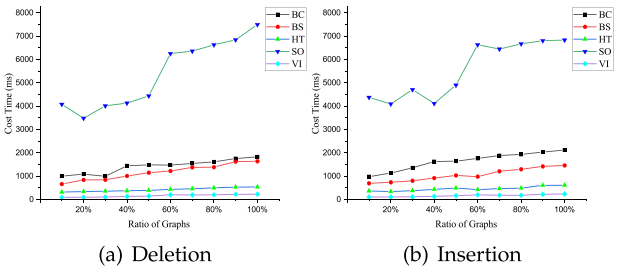


Fig. 8. Influence of the size of hypergraph.

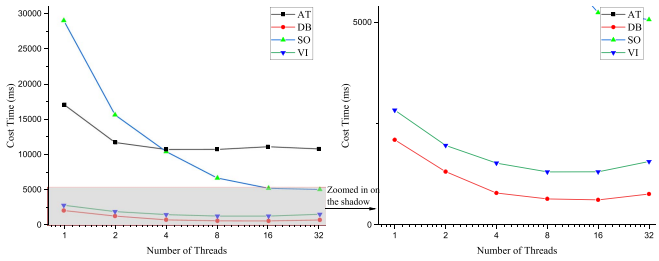


Fig. 9. Influence of the number of threads (deletion).

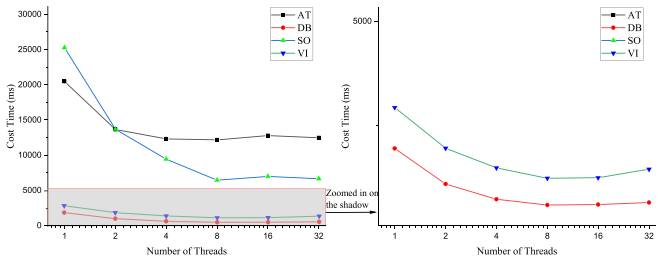


Fig. 10. Influence of the number of threads (insertion).

vertices are randomly deleted first and then inserted back for each sampled hyperedge. The number of threads is set to 1, 2, 4, 8, 16, 32 in turn. We record the cost time to maintain the core numbers under different threads. The experimental results are shown in Figs. 9 and 10. For deletion cases (Fig. 9), the cost time of the SO graph decreases as the number of threads increases. For the other three graphs, the cost time reaches the lowest point when the number of threads is 8 and then increases slightly with the increase of the number of threads. In the case of insertions (Fig. 10), the lowest point of the cost time for all four graphs, including the SO graph, also appears on 8 threads.

As shown in Fig. 3, the distributions of the core numbers of the hypergraphs are not uniform. The vertices with smaller core numbers account for a more significant proportion, which is the root cause of the limitation of parallelism. Threads corresponding to smaller core numbers tend to need to access more vertices and thus spend more time. Other threads may have completed their work while the threads corresponding to smaller core numbers are still working. Therefore, as the number of threads increases continually, there is no significant acceleration effect.

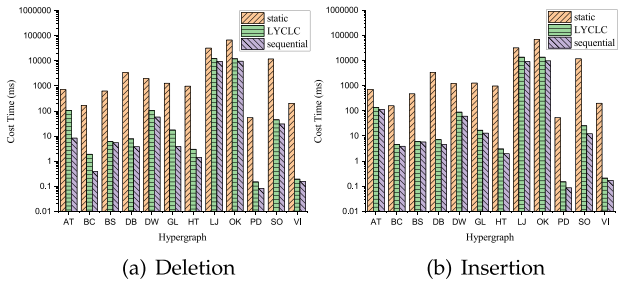


Fig. 11. Comparison with baseline algorithms for a single hyperedge.

D. Comparisons With Existing Algorithms

In this subsection, we first compare our sequential algorithm with the sequential static algorithms [12] and the LYCLC algorithms [13], and then compare our parallel algorithm with the parallel static algorithm [29] and the parallel dynamic GPC algorithm [29]. The two static algorithms are core decomposition of the entire hypergraph from scratch after the hypergraph changes. The LYCLC algorithms can effectively maintain core numbers after a single hyperedge is inserted/deleted. The GPC algorithm can maintain core numbers after the vertices in hyperedges change.

We randomly sample 1,000 hyperedges from six static hypergraphs and select the latest 1,000 hyperedges from six temporal hypergraphs according to timestamps. We randomly select several vertices from each hyperedge and delete them before inserting them. Our sequential algorithms are used to handle the insertion/deletion of vertices in a single hyperedge. The cost time of core maintenance is recorded after the algorithms are completed for each hyperedge. The average cost time of 1,000 hyperedges is taken as the final result. The result is shown in Fig. 11 where the cost time is in a log axis. Our sequential algorithms have the best performance in both deletion and insertion cases. Compared with static algorithms, the sequential algorithms have the highest speedup more than 1,000× (VI graph) and the average speedup more than 300×. Compared with the LYCLC algorithms, the sequential algorithms have the highest speedup more than 12× (the deletion of AT graph) and the average speedup 2.8×.

The above comparisons are for a single hyperedge. Next, we compare our parallel algorithms with algorithms for multiple hyperedges. The updated sets include 200, 400, 600, 800, 1,000 hyperedges. We randomly select several vertices from each hyperedge and delete them before inserting them. The deletion time and insertion time are recorded for our parallel algorithms, the parallel static algorithm and the parallel dynamic GPC algorithm. Our parallel algorithms divide these updated vertices into several matchings according to hyperedges they are on. Inserting/deleting all vertices from the same matching will only make core numbers change by 1 at most. All hyperedges contained by a matching are divided according to their core numbers. Each thread corresponds to a specific core number to speed up core maintenance.

The experimental results are shown in Fig. 12. We compare the average cost time each algorithm spent on an edge. The thread

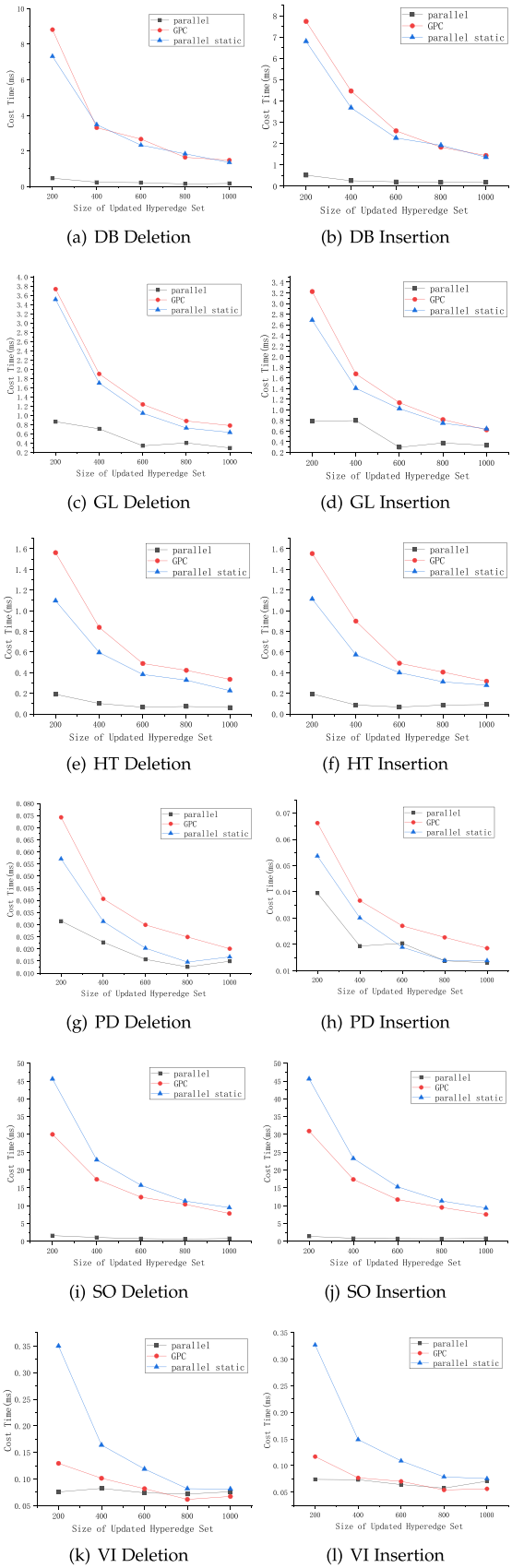


Fig. 12. Comparison with baseline algorithms for multi-hyperedges.

number is set to 32, where the parallel dynamic GPC algorithm performs the best. In general, since a larger updated set has a higher degree of parallelism, the average cost time decreases as the size of updated set increases. In some cases, the average cost time could be higher than others, such as the 800-hyperedges deletion and insertion of GL graph in Fig. 12(c) and (d), the 1,000-hyperedges deletion and 600-hyperedges insertion of PD graph in Fig. 12(g) and (h), and the 1,000-hyperedges insertion of VI graph in Fig. 12(l). It is because that the parallelism of our algorithms depends on matching divisions, and the distributions of core numbers of the hypergraphs are not uniform, thus a smaller random updated set could perform better.

It can be seen that our parallel algorithm performs better than other algorithms in most cases. The parallel dynamic GPC algorithm performs similarly to the parallel static algorithm, since they are both convergence based algorithms and the former one initialized a loose upper bound of temporary core numbers. On VI graph (Fig. 12(k) and (l)), our parallel algorithm spends more average cost time than the GPC algorithm when the size of updated set increases. The reason might be twofold. First, the size of VI graph is small and the cost time of each iteration in the convergence-based GPC algorithm on VI is less than on other graphs. Second, the size of updated set has more influences on our parallel algorithm than on the GPC algorithm since our parallel algorithm needs additional cost on matching divisions.

Compared with the parallel static algorithms, our parallel algorithms with 32 threads have the highest speedup more than 33× (the insertion of SO graph in Fig. 12(j)) and the average speedup around 7.3×. Compared with GPC algorithms, our algorithms have the highest speedup more than 22× (the insertion of SO graph in Fig. 12(j)) and the average speedup around 6.8×. Our parallel algorithms can reduce the redundant accesses of vertices and use multi-threading technology to speed up. Thus it has a significant advantage in handling the change of multiple hyperedges.

VI. Related Work

For a static unweighted and undirected graph, the straightforward algorithm to compute the core numbers is the peeling algorithm [19] that was proposed by Seidman. The peeling algorithm initializes the parameter t to one, removes all vertices whose degrees are less than or equal to t, and updates the degrees of their neighbors. If the degrees of these neighbors are less than or equal to t, they will also be removed. The core numbers of these removed vertices are t. Then, the peeling algorithm increases t by 1 and repeats the removal until there do not exist vertices in the graph. This algorithm can be implemented with a priority queue. Core decomposition and core maintenance in the ordinary graph have been widely studied. For the core decomposition of the static graphs, Batagelj and Zaversnik [3] designed the first algorithm with linear time complexity by using bin-sorting. Dasari, Ranjan, and Zubair [5] proposed the first parallel solution (the ParK algorithm) of core decomposition. Kabir and Madduri [11] designed a more scalable algorithm (PKC) based on ParK algorithm. In [4], Dhulipala, Blelloch, and

Shun represented the first work-efficient parallel core decomposition algorithm. Distributed core decomposition was discussed by Montresor, Pellegrini, and Miorandi [16]. Their algorithms utilized the idea of h-index. Aridhi et al. [2] designed distributed algorithms for core decomposition and core maintenance, which were implemented by using the AKKA framework.[3]

For the core maintenance in the dynamic graphs, Sariyüce et al. reported efficient sequential algorithms (the TRAVERSAL algorithms) to handle the insertion/deletion of a single edge [17]. Another work for a single edge was completed by Zhang et al. [28]. They designed k-order to reduce the redundant access of vertices so that their algorithms beat the TRAVERSAL algorithms. To deal with multiple edges in one iteration, Jin et al. [10], Wang et al. [25], and Hua et al. [9] proposed parallel algorithms based on different edge structures, respectively. Recently, Weng et al. [26] designed efficient algorithms of core maintenance for a distributed system. In addition, researchers studied many varieties of k-core, which can be obtained from the survey by Galimberti et al. [6] and Malliaros et al. [15].

Recently, k-core in the hypergraphs has attracted more and more attention. The core decomposition of static hypergraphs was first proposed by Leng et al. in [12]. Shun discussed a lot of parallel algorithms for hypergraphs in [21], including core decompositions by peeling. Gabert et al. [29] generalized the static core decomposition on ordinary graphs to hypergraphs by using a parallel algorithm based on convergence [31].

There are only a few studies focusing on core maintenance in dynamic hypergraphs. Sun et al. [22] considered core maintenance from an approximate view, which can update core numbers with a poly-logarithmic time complexity. Another work considering the exact core maintenance on dynamic hypergraphs was proposed by Luo et al. [13]. However, their algorithms aim to deal with the insertion/deletion of a single hyperedge. Gabert et al. also considered the exact core maintenance on dynamic hypergraphs. In [30], they generalized traversal and order algorithms to hypergraphs. Similar to work of [13], they did not consider the situation of vertices changes. In their another work [29], they presented parallel dynamic algorithms that supported changes to vertices.

VII. Conclusion

Existing works for core maintenance on dynamic hypergraphs mainly focus on the insertion/deletion of hyperedges, which means inserting/deleting all the vertices in those hyperedges. In this paper, we revisit this problem by observing that the hypergraphs can also be updated with insertion/deletion of certain vertices in particular hyperedges. Thus we study a generalized setting compared with previous works. We design both sequential and parallel algorithms to deal with the problem. Extensive experiment results show that our algorithms are superior to baseline algorithms.

In the future, we intend to optimize our parallel algorithms further to better deal with the insertion/deletion of vertices contained by multiple hyperedges. On one hand, we try to find better structures than a matching so that more hyperedges and vertices can be processed in each iteration. On the other hand, different from allocating threads according to the core numbers, other parallel schemes are also worth trying. In addition, maintaining other properties such as betweenness centrality after the change occurs is also an interesting topic in dynamic hypergraphs.

[3][Online]. Available: https://akka.io/

Acknowledgments

The authors thank Hongen Wang and Meng Wang for helping to implement the experiment.

References

[1] M. A. Al-garadi, K. D. Varathan, and S. D. Ravana, "Identification of influential spreaders in online social networks using interaction weighted k-core decomposition method," *Physica A: Statist. Mechanics Appl.*, vol. 468, pp. 278–288, 2017.
[2] S. Aridhi, M. Brugnara, A. Montresor, and Y. Velegrakis, "Distributed k-core decomposition and maintenance in large dynamic graphs," in *Proc. 10th ACM Int. Conf. Distrib. Event-Based Syst.*, 2016, pp. 161–168.
[3] V. Batagelj and M. Zaversnik, "An O(m) algorithm for cores decomposition of networks," 2003, *arXiv:0310049*.
[4] L. Dhulipala, G. E. Blelloch, and J. Shun, "Julienne: A framework for parallel graph algorithms using work-efficient bucketing," in *Proc. 29th ACM Symp. Parallelism Algorithms Architectures*, 2017, pp. 293–304.
[5] N. S. Dasari, D. Ranjan, and M. Zubair, "ParK: An efficient algorithm for k-core decomposition on multicore processors," in *Proc. IEEE Int. Conf. Big Data*, 2014, pp. 9–16.
[6] E. Galimberti, F. Bonchi, F. Gullo, and T. Lanciano, "Core decomposition in multilayer networks: Theory, algorithms, and applications," *ACM Trans. Knowl. Discov. Data*, vol. 14, no. 1, pp. 11:1–11:40, 2020.
[7] P. Govindan, C. Wang, C. Xu, H. Duan, and S. Soundarajan, "The k-peak decomposition: Mapping the global structure of graphs," in *Proc. 26th Int. Conf. World Wide Web*, 2017, pp. 1441–1450.
[8] L. Hébert-Dufresne, A. Allard, J.-G. Young, and L. J. Dubé, "Percolation on random networks with arbitrary k-core structure," *Phys. Rev. E*, vol. 88, no. 6, 2013, Art. no. 062820.
[9] Q.-S. Hua et al., "Faster parallel core maintenance algorithms in dynamic graphs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 6, pp. 1287–1300, Jun. 2020.
[10] 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 Trans. Parallel Distrib. Syst.*, vol. 29, no. 11, pp. 2416–2428, Nov. 2018.
[11] H. Kabir and K. Madduri, "Parallel k-core decomposition on multicore platforms," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2017, pp. 1482–1491.
[12] M. Leng, L. Sun, J. Bian, and Y. Ma, "An O(m) algorithm for cores decomposition of undirected hypergraph," *J. Chin. Comput. Syst.*, vol. 34, no. 11, pp. 2568–2573, 2013.
[13] Q. Luo, D. Yu, Z. Cai, X. Lin, and X. Cheng, "Hypercore maintenance in dynamic hypergraphs," in *Proc. IEEE 37th Int. Conf. Data Eng.*, 2021, pp. 2051–2056.
[14] Q. Luo, D. Yu, X. Cheng, Z. Cai, J. Yu, and W. Lv, "Batch processing for truss maintenance in large dynamic graphs," *IEEE Trans. Comput. Soc. Syst.*, vol. 7, no. 6, pp. 1435–1446, Dec. 2020.
[15] F. D. Malliaros, C. Giatsidis, A. N. Papadopoulos, and M. Vazirgiannis, "The core decomposition of networks: Theory, algorithms and applications," *VLDB J.*, vol. 29, no. 1, pp. 61–92, 2020.
[16] A. Montresor, F. D. Pellegrini, and D. Miorandi, "Distributed k-core decomposition," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 2, pp. 288–300, Feb. 2013.
[17] A. E. Sariyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek, "Streaming algorithms for k-core decomposition," *Proc. VLDB Endowment*, vol. 6, no. 6, pp. 433–444, 2013.
[18] A. E. Sariyüce, C. Seshadhri, A. Pinar, and Ü. V. Çatalyürek, "Finding the hierarchy of dense subgraphs using nucleus decompositions," in *Proc. 24th Int. Conf. World Wide Web*, 2015, pp. 927–937.
[19] S. B. Seidman, "Network structure and minimum degree," *Soc. Netw.*, vol. 5, pp. 269–287, 1983.

[20] K. Shin, T. Eliassi-Rad, and C. Faloutsos, "CoreScope: Graph mining using k-core analysis - patterns, anomalies and algorithms," in *Proc. IEEE 16th Int. Conf. Data Mining*, 2016, pp. 469–478.
[21] J. Shun, "Practical parallel hypergraph algorithms," in *Proc. 25th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2020, pp. 232–249.
[22] B. Sun, T.-H. H. Chan, and M. Sozio, "Fully dynamic approximate k-core decomposition in hypergraphs," *ACM Trans. Knowl. Discov. Data*, vol. 14, no. 4, pp. 39:1–39:21, 2020.
[23] A. J.-P. Tixier, K. Skianis, and M. Vazirgiannis, "GoWvis: A web application for graph-of-words-based text visualization and summarization," in *Proc. ACL Syst. Demonstrations*, 2016, pp. 151–156.
[24] J. Wang and J. Cheng, "Truss decomposition in massive networks," *Proc. VLDB Endowment*, vol. 5, no. 9, pp. 812–823, 2012.
[25] N. Wang, D. Yu, H. Jin, C. Qian, X. Xie, and Q.-S. Hua, "Parallel algorithm for core maintenance in dynamic graphs," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst.*, 2017, pp. 2366–2371.
[26] T. Weng, X. Zhou, K. Li, P. Peng, and K. Li, "Efficient distributed approaches to core maintenance on large dynamic graphs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 1, pp. 129–143, Jan. 2022.
[27] Y. Zhang and S. Parthasarathy, "Extracting analyzing and visualizing triangle k-core motifs within networks," in *Proc. IEEE 28th Int. Conf. Data Eng.*, 2012, pp. 1049–1060.
[28] Y. Zhang, J. X. Yu, Y. Zhang, and L. Qin, "A fast order-based approach for core maintenance," in *Proc. IEEE 33rd Int. Conf. Data Eng.*, 2017, pp. 337–348.
[29] K. Gabert, A. Pinar, and Ü. V. Çatalyürek, "Shared-memory scalable k-core maintenance on dynamic graphs and hypergraphs," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2021, pp. 998–1007.
[30] K. Gabert, A. Pinar, and Ü. V. Çatalyürek, "A unifying framework to identify dense subgraphs on streams: Graph nuclei to hypergraph cores," in *Proc. 14th ACM Int. Conf. Web Search Data Mining*, 2021, pp. 689–697.
[31] L. Lü et al., "The h-index of a network node and its relation to degree and coreness," *Nature Commun.*, vol. 7, no. 1, pp. 1–7, 2016.

Qiang-Sheng Hua (Member, IEEE) received the BEng and MEng degrees from the School of Computer Science and Engineering, Central South University, China, in 2001 and 2004, respectively, and the PhD degree from the Department of Computer Science, The University of Hong Kong, China, in 2009. He is currently a professor with the Huazhong University of Science and Technology, China. He is interested in the algorithmic aspects of parallel and distributed computing.

Xiaohui Zhang received the BE and ME degrees from the Huazhong University of Science and Technology, in 2019 and 2022, respectively. His research interests include dynamic graph algorithms and parallel computing.

Hai Jin (Fellow, IEEE) received the PhD degree from the Huazhong University of Science and Technology (HUST), in 1994. He is a professor with the School of Computer Science and Technology, Huazhong University of Science and Technology. He was a postdoctoral fellow with the University of Southern California and The University of Hong Kong. His research interests include HPC, grid computing, cloud computing, and virtualization.

Hong Huang (Member, IEEE) received the ME degree in electronic engineering from Tsinghua University, Beijing, China, in 2012, and the PhD degree in computer science from the University of Göttingen, Germany, in 2016. She is an associate professor with the Huazhong University of Science and Technology, China. Her research interests lie in social network analysis, data mining and knowledge graph.