Faster Parallel Core Maintenance Algorithms in Dynamic Graphs

Qiang-Sheng Hua, *Member, IEEE*, Yuliang Shi, Dongxiao Yu, *Member, IEEE*, Hai Jin, *Fellow, IEEE*, Jiguo Yu, *Senior Member, IEEE*, Zhipen Cai, *Member, IEEE*, Xiuzhen Cheng, *Fellow, IEEE*, and Hanhua Chen, *Member, IEEE*

Abstract—This article studies the core maintenance problem for dynamic graphs which requires to update each vertex's core number with the insertion/deletion of vertices/edges. Previous algorithms can either process one edge associated with a vertex in each iteration or can only process one superior edge associated with the vertex (an edge $\langle u, v\rangle$ is a superior edge of vertex u if v' core number is no less than u's core number) in each iteration. Thus for high superior-degree vertices (the vertices associated with many superior edges) insertions/deletions, previous algorithms become very inefficient. In this article, we discovered a new structure called joint edge set whose insertions/deletions make each vertex's core number change at most one. The joint edge set mainly contains all the superior edges associated with the high superior-degree vertices as long as these vertices are 3^+-hop independent. Based on this discovery, faster parallel algorithms are devised to solve the core maintenance problems. In our algorithms, we can process all edges in the joint edge set in one iteration and thus can greatly increase the parallelism and reduce the processing time. The results of extensive experiments conducted on various types of real-world, temporal, and synthetic graphs illustrate that the proposed algorithms achieve good efficiency, stability and scalability. Specifically, the new algorithms can outperform the single-edge processing algorithms by up to four orders of magnitude. Compared with the matching based algorithm and the superior edge based algorithm, our algorithms show a significant speedup up to 60× in the processing time.

Index Terms—Graph analysis, core maintenance problem, parallel algorithm

1 Introduction

Core number is one of the most efficient and helpful indexes adopted in graph analytics to depict the cohesiveness of a graph, as previous work has proposed a linear time complexity algorithm to compute this index defined on vertices of a graph [6]. Specifically, the k-core in graph G is a subgraph that each vertex's degree in the subgraph is no less than k. For a given vertex v, if we can find that v is contained in a k-core and cannot find v is contained in a $(k+1)$-core, we say vertex v's core number equals to k. Core number has been widely used in large numbers of real-world applications, including analyzing the Internet topology [8], learning dynamic dependency network structure [12], the study of the influential spreader in complex networks [17], large-scale software systems structure analysis [19], [24], the prediction of the function of biology networks [4], and graph visualization [2].

As dynamicity is inherent in a wide spectrum of graph applications, such as social networks where there are persistent node joining/leaving or edge insertion/deletion. Recent studies on core computation turn attention to the core maintenance problem, which is to correctly update each vertex's core number after the graph change. If we recompute the vertices' core numbers using the algorithms described in static graphs, the time and space cost is high especially in large-scale graphs with millions of vertices and hundreds of millions of edges. Besides, the number of updated edges is usually small such that only a small proportion of vertices need to update their core numbers. Thus, the two main issues to be solved in core maintenance as discussed in [14] are: (1) find the vertices whose core numbers will change after the insertion/deletion of edges, and (2) identify how large the core numbers of these vertices have changed. The main difficulties are, even if the same number of edges are inserted/deleted, the vertex set whose core numbers will change and the changed values may be different.

To overcome the challenges for core maintenance, Sariyüce *et al.* [23] presented the single edge based algorithm focusing on the case of one edge insertion/deletion in

- *Q.-S. Hua, Y. Shi, H. Jin, and H. Chen are with 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, P.R. China. E-mail: {qshua, m201672783, hjin, chen}@hust.edu.cn.*
- *D. Yu and X. Cheng are with the School of Computer Science and Technology, Shandong University, Qingdao 266237, P.R. China. E-mail: {dxyu, xzcheng}@sdu.edu.cn.*
- *J. Yu is with the School of Computer Science and Technology (Shandong Academy of Sciences), Jinan, Shandong, 250353, P.R. China. E-mail: jiguoyu@sina.com.*
- *Z. Cai is with the Department of Computer Science, Georgia State University, Atlanta, GA 30303. E-mail: zcai@gsu.edu.*

Manuscript received 24 Apr. 2019; revised 28 Oct. 2019; accepted 10 Dec. 2019. Date of publication 17 Dec. 2019; date of current version 27 Jan. 2020. (Corresponding author: Dongxiao Yu.)
Recommended for acceptance by M. Guo.
Digital Object Identifier no. 10.1109/TPDS.2019.2960226

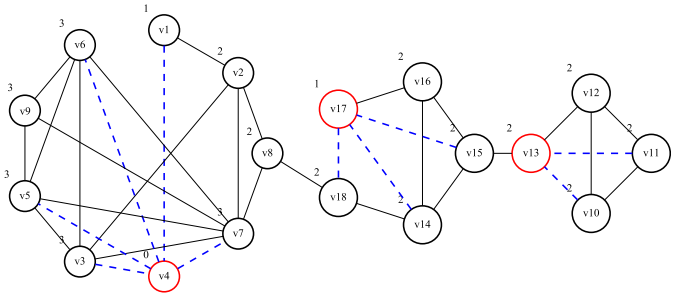
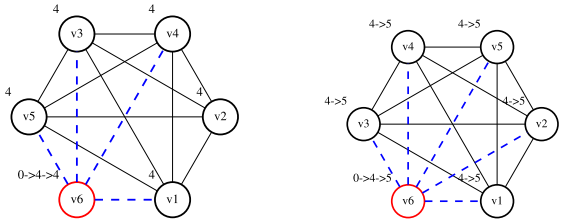


Fig. 1. In the sample graph G, the dotted lines are newly added edges. The number on top of each vertex means the vertex's core number before inserted edges. Vertex v_4 is a newly added vertex with initial core number 0 and thus all its connected edges are superior edges. The added edges for existing vertices v_{13}, v_{17} are also superior edges since they connect to higher or equal core number vertices.



(a) The pre-core number of v_6 is 4 and the final core number is 4.

(b) The pre-core number of v_6 is 4 and the final core number is 5.

Fig. 2. In (a) and (b), new vertex v_6 joins the original graph and adds some edges to existing vertices. The final core number the vertex v_6 is either equal to pre-core(v_6) or pre-core(v_6)+1.

each iteration. They proved that in this scenario, each vertex's core number will be changed by at most 1. Hence, it overcomes the second issue as above mentioned. As a result, it only needs to find an efficient algorithm to identify the vertex set that each vertex in the set changes its core number. Scott *et al.* [21] presented incremental graph processing for on-line analytics. Based on the single edge algorithm, Jin *et al.* [14] presented a matching based algorithm and Wang *et al.* [25] presented a superior edge based algorithm to simultaneously insert/delete multiple edges and ensure that each vertex's core number change is at most 1. Thus, more edges can be processed in each iteration and the total number of iterations can be reduced.

However, the matching based algorithm can only process one edge associated to a vertex in each iteration and the superior edge based algorithm can only process one superior edge associated to the vertex ($\langle u, v\rangle$ is a superior edge for vertex u if v's core number is no less than u's core number) in each iteration. When considering the scenario that a vertex v adds multiple superior edges, for example the vertex v_4 in Fig. 1 adds 5 superior edges, the algorithms in [14] and [25] need at least 5 iterations to process the newly added edges. Thus we need to give special treatment to these vertices to reduce the iteration times.

In this paper, we prove that: when we insert/delete all superior edges associated to the vertex v in one iteration, each other vertex's core number can be changed by at most 1 from its previous core number; and the vertex v's core number can be changed by at most 1 from its defined pre-core number(c.f. Definition 7 and Fig. 2) which is calculated from v's all neighbors' previous core numbers.

Based on this observation, in this paper, a new algorithm is presented for core maintenance that admits multiple edges insertions/deletions in one iteration. In particular, we treat those vertices inserting/deleting multiple superior edges as *central vertices*, and find an *available structure* of inserted/deleted edges that the insertion/deletion of the structure makes each vertex except central vertices change the core number by at most 1. As for each vertex in central vertices, its core number changes by at most 1 from its pre-core number. Then the core maintenance problem can also be simplified as finding the vertex set that each vertex in the set will change its core number with the insertion/deletion of the available structure. As a result, we can devise efficient core maintenance algorithms consisting of two key steps in each iteration: 1) find the available structure, and 2) find out those vertices whose core numbers really change with the insertion/deletion of the available structure.

The following summarizes the major contributions:

- We propose an available structure called *Joint Edge Set* whose insertion/deletion in each iteration makes each vertex except the central vertices change its core number by at most 1.
- Based on the proposed joint edge set structure and by adapting the TRAVERSAL algorithms for single-edge insertion/deletion in [23], we present parallel core maintenance algorithms that can handle multiple edge insertions/deletions simultaneously. Our algorithms can greatly reduce the times of iterations needed for core number update. If we denote the number of inserted/deleted edges as m_c, the sequential single-edge processing algorithm needs m_c iterations, while our algorithms can reduce the times of iterations to $\min\{\Delta_2, \Delta_e\}$, where Δ_2 is the maximum number of central vertices within each central vertex's 2-hop neighbors, and Δ_e is the maximum number of edges inserted/deleted to every vertex. Notice that even if a large number of edges are inserted/deleted, Δ_2 and Δ_e are usually very small in large-scale networks. Compared with algorithms in [25] and [14], our special treatment to central vertices can greatly reduce iteration times with the insertion/deletion of multiple vertices/edges.
- Extensive experiments conducted on various kinds of graphs, including real-world, temporal and synthetic graphs show that the proposed algorithms achieve good stability and scalability. Besides, when comparing with existing algorithms, our approach outperforms the single-edge approach by Sariyüce [23] by up to four orders of magnitude. In addition, our algorithms can achieve up to 60 $\times$ faster than the matching based algorithm in [14] (Fig. 7a) and the superior edge based algorithm in [25] (Fig. 8a), especially when there are lots of vertices/edges insertions/deletions.

The rest of this paper is organized as follows. We briefly discuss the related work in Section 2 and give some formal definitions used in this paper in Section 3. The theoretical basis of our algorithms is given in Section 4. Based on the theoretical analysis, we propose the parallel core maintenance algorithms for the insertion and deletion cases in Sections 5 and 6, respectively. We conduct extensive experiments and

present the results in Section 7. We show the algorithm's application in distributed core decomposition in Section 8. The conclusion of our work is made in Section 9.

2 Related Work

The core decomposition problem is to compute the core number of each vertex in static graphs. Batagelj and Zaversnik [6] presented a linear time complexity algorithm to compute the core numbers. It is a bottom-up approach to continuously remove vertices whose degrees are less than k until all vertices are processed. In [10], Cheng *et al.* proposed a disk oriented algorithm when the random access memory is too small to hold the entire graph. Their top-down approach computes core numbers from large values to smaller ones recursively. It greatly reduces the disk I/O cost. Based on the locality property of the core decomposition, Montresor, Pellegrini and Miorandi studied the distributed k-core decomposition in [20]. The above three algorithms were implemented and compared in [16] under the GraphChi and WebGraph models using a single consumer-grade machine. Besides, with the popularity of multi-core processors, Dasari, Desh and Zubair proposed a parallel core decomposition algorithm in [11]. Their experiment results indicate that the parallel algorithm using 32 cores can achieve speedup up to 21 times compared with the sequential algorithm.

In contrast, the core maintenance problem is to update a subset of vertices' core numbers instead of recomputing with the evolvement of the graph. Specifically, in [23], Sariyüce *et al.* proved that the core number change of each vertex is at most one after inserting/deleting one edge. Based on this statement, they proposed a fast algorithm called *TRAVERSE* to identify the vertex set whose core numbers really change with the insertion/deletion of edges. In [18], Li, Yu and Mao gave a similar result only to maintain certain vertices' core numbers when the graph insert/delete an edge. In [26], Wen *et al.* proposed a disk oriented approach for I/O efficient core decomposition. Distributed algorithms for the core maintenance were studied in [1] and [3]. They both aggressively prune unnecessary computations and only need to re-evaluate the core number of a fixed number of vertices after the edge insertions/deletions. In [25] and [14], Wang *et al.* proposed parallel algorithms to simultaneously process multiple edges in each iteration. They are the first one to explore the available structure that the insertion/deletion of the structure can only make each vertex's core number change by at most 1.

3 Problem Definitions

Given an undirected and unweighted graph $G = (V, E)$, where $V(G)$ and $E(G)$ represent the sets of vertices and edges in G, respectively. Let $N = |V(G)|$ and $M = |E(G)|$. In this paper, for a vertex $u \in V(G)$, u's neighbors and degree are denoted as $N_G(u)$ and $d_G(u) = |N_G(u)|$, respectively. When the context is clear, they are simplified as $N(u)$ and $d(u)$, respectively. $\Delta(G)$ is the maximum degree of vertices in G. $\delta(G)$ is the minimum degree of vertices in G. We say graph H is a subgraph of G if it satisfies that $E(H) \subseteq E(G)$ and $V(H) \subseteq V(G)$. We next give some useful formal definitions.

Definition 1 (k-Core). *A k-core of a graph G is a maximal subgraph H of G that $\forall v \in V(H), d_H(v) \geq k$, i.e., $\delta(H) \geq k$.*

Definition 2 (Max-k-Core). *For a given vertex v in the graph G, if v is contained in a k-core and there does not exist a $(k+1)$-core containing vertex v, the k-core is called the max-k-core of vertex v in graph G.*

Definition 3 (Core Number of a Vertex). *For a given vertex v in the graph G, if there exists a max-k-core containing vertex v, the core number of vertex v equals to k. We use $core_G(v)$ to denote the vertex v's core number in graph G. When the context is clear, it can be simplified as $core(v)$.*

According to the above definitions, we have the following equation to compute the core number of the vertex v

$$core(v) = arg\max_{K \geq 0}\{|\{u \in N(v) | core(u) \geq K\}| \geq K\}. \tag{1}$$

The equation tells us that one vertex's core number is related only to its neighbors' core numbers information.

Definition 4 (Core Number of an Edge). *For an edge $e = \langle u, v\rangle$, the* core number *of the edge e equals to the smaller value of $core(u)$ and $core(v)$. We use $core(e)$ to denote edge e's core number.*

Definition 5 (Superior Edge[25]). *For an edge $e = \langle u, v\rangle$ in graph G, if $core(u) \leq core(v)$, we say edge e is a superior edge of vertex u.*

Definition 6 (Superior Degree). *For a graph G, the superior degree of v is defined as the number of superior edges of v. We use $SD_G(v)$ to denote vertex v's superior degree. When the context is clear, it can be simplified as $SD(v)$.*

In this work, we study the core maintenance problems with the insertions/deletions of vertices/edges in dynamic graphs. Specifically, the core maintenance problems under the insertion and the deletion cases are known as the *incremental* and the *decremental* core maintenance, respectively.

4 Theoretical Basis

As mentioned earlier, the state-of-art algorithms can either process one edge associated to a vertex in each iteration (the matching based method in [14]) or can only process one superior edge associated to the vertex in each iteration (the superior edge based method in [25]). Thus, for high superior-degree vertices (the vertices with many superior edges) insertions/deletions, these algorithms become inefficient.

We discover that if we compute the pre-core values (Definition 7) of the vertices in a 3^+-hop independent set (Definition 8) in advance, each vertex in the independent set can change its core number by at most 1 from its pre-core value. Thus, for each vertex in the 3^+-hop independent set, even it adds/deletes arbitrary number of superior edges, we can process these edges in one iteration. Based on this discovery, we propose a new structure called joint edge set whose insertions/deletions only make each vertex change its core number by at most 1. The joint edge set mainly contains all the edges with the high superior-degree vertices as long as these vertices are 3^+-hop independent. Note that the matching structure

TABLE 1
Notations and Their Descriptions

Notations	Description
$N_G(u)$	vertex u's neighbors in graph G
$d_G(u)$	vertex u's degree in graph G
$\Delta(G)$	the maximum degree of vertices in graph G
$\delta(G)$	the minimum degree of vertices in graph G
$core_G(u)$	vertex u's core number in graph G(c.f. Definition 3)
$SD_G(u)$	vertex u's superior degree in graph G(c.f. Definition 6)
$pre\text{-}core(u)$	vertex u's pre-core number(c.f. Definition 7)
V_c	central vertex set (c.f. Definition 11)
KPT_u	K-Path-Tree of vertex u(c.f. Definition 15)
R_I	insertion root vertex set(c.f. Definition 16)
R_D	deletion root vertex set(c.f. Definition 17)

in [14] is a subset of the superior edge structure in [25], and the superior edge structure is a subset of the proposed joint edge set structure. Table 1 lists some important notations with their descriptions used in this paper.

4.1 The New Structure Called Joint Edge Set

After the graph change, the core numbers of vertices computed using Equation (1) may not be correct any more. But as shown later, this value is very helpful in the core number update procedure. Hence, we define this value as the $pre\text{-}core$ of each vertex v in the changed graph, denoted as $pre\text{-}core(v)$. The formal definition is given as follows.

Definition 7. *After inserting/deleting edges into graph $G = (V, E)$, G becomes $G' = (V', E')$. The* pre-core *number of the vertex $v \in V'$ is defined as $arg\max_{K\geq 0}\{|\{u \in N_{G'}(v)|core_G(u) \geq K\}| \geq K\}$.*

For example, considering the vertex v_4 in Fig. 1, it adds 5 new neighbors whose core numbers are core(v_1)=1, core(v_3) =3, core(v_5)=3, core(v_6)=3 and core(v_7)=3. According to Definition 7, $K = 3$ is the maximum value satisfying the condition. Thus, pre-core(v_4)=3.

We next give two sufficient conditions for the core number change, which will be useful in the subsequent proofs.

Lemma 1. *Considering a vertex $w \in V$ and $core(w) = q$, if the core number of each vertex in w's neighbors increases/decreases by at most 1, then the vertex w's core number will increase/decrease by at most 1.*

The proof of Lemma 1 can be found in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2019.2960226.

Lemma 2. *After the insertion/deletion of a vertex u and its connected edges $\langle u, v_1\rangle, \langle u, v_2\rangle, \ldots, \langle u, v_p\rangle$ in graph G, for each vertex $w \in V(G), w \neq u$ and $core(w) = q$, its core number changes by at most 1.*

Proof. We first consider the insertion case. We assume that vertex w's core number increases by $\Delta q > 1$ to $q + \Delta q$ after the insertion of edges. The max-k-core of vertex w before the insertion is denoted as H^w. The max-k-core of vertex w after the insertion is denoted as H^w_+. It can be concluded that $u \in H^w_+$, as otherwise H^w_+ is a $(q + \Delta q)$-core before the insertion, which conflicts with our assumption. As $core(w) = q + \Delta q$ after the insertion of edges, w has at least $q + \Delta q$ neighbors in H^w_+ whose degrees are at least $q + \Delta q$. Let $Z = H^w_+ \setminus u$, if Z is connected, we can find that w has at least $q + \Delta q$-1 neighbors whose degrees are at least $q + \Delta q - 1$, since the removal of a single vertex can make the degree of every vertex in Z decrease by at most 1. As a result, Z must form a $(q + \Delta q - 1)$-core, which conflicts with our assumption, since $q + \Delta q - 1 > q$ but H^w is the maximal k-core before the insertion. If Z is disconnected, there are at most $p + 1$ components as we insert p edges. We denote the components as C_i where $1 \leq i \leq p + 1$. In H^w_+ each vertex has at least $q + \Delta q$ neighbors, as each vertex's degree reduces by at most 1 after removing vertex u, there are at least $q + \Delta q - 1$ neighbors for each vertex in graph Z. So each component C_i must be a $(q + \Delta q - 1)$-core and vertex w is contained in one of these components, which also leads to a contradiction.

Next, we consider the deletion case. We assume that $core(w)$ is decreased by $\Delta q > 1$ after the removal of vertex u. If we add vertex u back to the graph, vertex w's core number will increase by Δq, which contradicts with the result proved above in the insertion case. □

By the above Lemma 2, after the insertion/deletion of a vertex u and its connected edges, the core number change of each vertex except u is at most 1. For vertex u itself, by Lemma 1, its core number change is at most 1 from pre-core(u), since the core number changes of all its neighbors are at most 1. As pre-core(u) can be calculated using all its neighbors' core numbers in the original graph, to simplify our presentation, when we say vertex u changes its core number by at most 1, we mean it changes by at most 1 from pre-core(u).

Then we reconsider the situation when a new vertex joins the original graph. As shown in Figs. 2a and 2b, we can calculate the pre-core number of the newly added vertex in advance and need only one another iteration to identify the final core numbers of all vertices. This approach is more efficient than the algorithms given in [25] and [14] since they need at least m' iterations, where m' is the number of the new inserted superior edges the newly added vertex connected to.

In fact, it can be ensured that even if multiple vertices are deleted from/inserted into a graph, every other vertex's core number will be changed by at most 1, as long as these deleted/inserted vertices satisfy some properties as defined subsequently. Before that, we first introduce some notation.

Definition 8 (3$^+$-hop Independent Set). *For a graph G, the 3$^+$-hop independent set of $V(G)$, denoted as V_{3h}, is a subset of $V(G)$ in which for any u, v, $N_G[u] \cap N_G[v] = \emptyset$.*

Definition 9 (Maximal 3$^+$-hop Independent Set). *For a graph G, V_{3h} is a 3$^+$-hop independent set in G. If V_{3h} is maximal, i.e., there does not exist a vertex $u \in V(G), u \notin V_{3h}$ such that $\forall v \in V_{3h}$, $N_G[u] \cap N_G[v] = \emptyset$, we say V_{3h} is a maximal 3$^+$-hop independent set of $V(G)$ in G.*

We then consider the insertion/deletion of multiple vertices.

Lemma 3. *Given a graph $G = (V, E)$, a vertex set $V' = \{u_1, u_2, \ldots, u_p\}$ and its connected edge set $E' = \{e_1, e_2, \ldots, e_k\}$, after*

inserting/deleting E' into/from G, G becomes G'. If it satisfies that V' constitutes a 3^+-hop independent set in the new graph, that is to say, $N_{G'}[u_i] \cap N_{G'}[u_j] = \emptyset$, where $1 \leq i \leq p$, $1 \leq j \leq p$, $i \neq j$, then for each vertex $u \in V \setminus V'$, $core(u)$ can change by at most 1.

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

By the above Lemmas 3 and 1, it can be obtained that with the insertion/deletion of a 3^+-hop independent vertex V_{3h} from a graph, each vertex $u \in V_{3h}$ changes its core number by at most 1 from $pre\text{-}core(u)$. Hence even if a large number of edges are inserted/deleted due to the insertion/deletion of a 3^+-hop independent set, all these edges can be handled efficiently in one iteration.

Definition 10 (Superior Vertex Set). *For a graph G, we insert/delete edges E' into/from the graph. V' is the vertex set connected to E'. For a vertex $u \in V'$, if edge $e = \langle u, v\rangle \in E'$ and $core_G(v) \geq core_G(u)$, edge e is a* newly inserted/deleted superior edge *of u. Superior vertex set contains vertex $u \in V'$ such that the number of u's newly inserted/deleted superior edges is more than 1.*

For example, in Fig. 1, $\{v_4, v_{13}, v_{17}\}$ is a superior vertex set as each vertex adds at least two superior edges.

Definition 11 (Central Vertex Set). *With the insertion/deletion of an edge set E', graph G becomes G'. The central vertex set in graph G' is a subset of the superior vertex set, denoted as V_c, which satisfies a maximal 3^+-hop independent set in graph G'. Each vertex in V_c is called a central vertex.*

For example, in Fig. 1, the sets $\{v_4, v_{13}\}$ and $\{v_4, v_{17}\}$ are both central vertex sets as each vertex in the set shares no common neighbors with other vertex in the set. However, $\{v_{13}, v_{17}\}$ is not a central vertex set as v_{13} and v_{17} have a common neighbor v_{15} in the updated graph.

By Lemmas 3 and 1, it has been shown that the edges connected to vertices in a central vertex set can be processed together. A question is whether it is possible to handle more edges. In the following, we define a new structure of inserted/deleted edges, called *Joint Edge Set* (JES), based on edges connected to the central vertex set and the *superior edge set* structure presented in [25] whose insertion/deletion also ensures that each vertex's core number will be changed by at most 1 from its previous core number. We next simply borrow the definition of the superior edge set.

Definition 12 (k-Superior Edge Set [25]). *A k-superior edge set $E_k = \{e_1, e_2, \ldots, e_p\}$ satisfies that if one edge (with core number k) $e_i = \langle u, v\rangle \in E_k$ and $core(u) \leq core(v)$, there does not exist another edge $e_j = \langle u, w\rangle \in E_k, i \neq j$ and $core(u) \leq core(w)$.*

It can be known that there does not exist one vertex in the k-superior edge set connecting to more than two superior edges. For example, in Fig. 1, each edge connected to vertex v_4 constitutes a 0-superior edge set and each of these five 0-superior edge sets can only have one edge in this example. In [25], it is shown that only vertices whose core numbers equal to k will be affected with the insertion/deletion of a k-superior edge set.

Lemma 4 ([25]). *With the insertion/deletion of a k-superior edge set, only the vertices whose core numbers equal to k will change by at most 1 and other vertices whose core numbers greater or smaller than k will not be affected.*

Further, a *superior edge set* is the union of all distinct k-superior edge sets. Then we have the following Lemma 5.

Lemma 5 ([25]). *After the insertion/deletion of a superior edge set $E_{k_1} \cup E_{k_2} \cup \cdots \cup E_{k_q}$ in the graph G, where $E_{k_i}, 1 \leq i \leq q$ is a k_i superior edge set, the core number of each vertex in $V(G)$ will be changed by at most 1.*

Based on the above discussions, we can define our new structure called joint edge set.

Definition 13 (Joint Edge Set (JES)). *Given a graph G, after inserting/deleting edges E', G becomes G', and V_c is a central vertex set of G'. The JES contains the following two types of edges in E':*

- *(i) an edge $e_i = \langle u, v\rangle \in E', u \in V_c$ or $v \in V_c$, then $e_i \in$ JES. We denote the set of these edges as E_c;*
- *(ii) assume edge $e_j = \langle u, v\rangle \in E' \setminus E_c$ is a superior edge of u, and u does not have another superior edge contained in JES, then $e_j \in$ JES. We denote the set of these edges as E_s.*

The matching structure in [13] is a subset of the superior edge structure in [22], and the superior edge structure is a subset of the proposed joint edge set structure of this paper.

For example, in Fig. 1, $V_c = \{v_4, v_{17}\}$ is a central vertex set, E_c contains all inserted/deleted edges associated to each vertex in V_c, i.e., the 8 edges set $\{\langle v_4, v_3\rangle, \langle v_4, v_5\rangle, \langle v_4, v_6\rangle, \langle v_4, v_1\rangle, \langle v_4, v_7\rangle, \langle v_{17}, v_{18}\rangle, \langle v_{17}, v_{14}\rangle, \langle v_{17}, v_{15}\rangle\}$, and $E_s = \{\langle v_{13}, v_{11}\rangle\}$ or $\{\langle v_{13}, v_{10}\rangle\}$.

4.2 Core Number Change of Vertices Affected by Joint Edge Set

We next show that with the insertion/deletion of a joint edge set, each vertex's core number will be changed by at most 1 from its previous core number (for vertex not in V_c) or from its pre-core (for vertex in V_c).

Because a joint edge set only contains two types of edges: E_c and E_s, according to Lemmas 3 and 5, we can get the following Lemma 6.

Lemma 6. *After the insertion of a joint edge set $E_U = E_c \cup E_s$ in the graph G, $\forall u \in V(G) \setminus V_c$, $core(u)$ can increase by at most 2.*

Proof. It can be seen that inserting the joint edge set has the same result as the scenario of inserting E_s first and then inserting E_c. According to Lemma 5, each vertex in $V(G) \setminus V_c$ will increase its core number by at most 1 after inserting E_s. Then we insert E_c. It can be seen that V_c is a 3^+-hop independent set as no edges in E_s connect to vertices in V_c. According to Lemma 3, each vertex in $V(G) \setminus V_c$ will increase its core number by at most 1. In summary, $\forall u \in V(G) \setminus V_c$, $core(u)$ can increase by at most 2. □

The result in Lemma 6 can be further optimized as shown later that the core number change upper bound can be reduced from 2 to 1. Before that, we partition the edges in a joint edge set into different groups and define each group as a k-joint edge set.

Definition 14 (k-Joint Edge Set). *Given a joint edge set $E_U = E_c \cup E_s$, an edge set $E_k = \{e_1, e_2, \ldots, e_p\} \subseteq (E_c \cup E_s)$ is a k-joint edge set if each edge $e_i = \langle u_i, v_i \rangle$, $1 \leq i \leq p$ satisfies one of the following two conditions:*

(i) $e_i \in E_c$, $u_i \in V_c$ *and* $core(v_i) = k$.
(ii) $e_i \in E_s$ *and the core number of* e_i *equals to* k.

A joint edge set $E_c \cup E_s$ consists of the union of all distinct k-joint edge sets. We first show that the insertion of a k-joint edge set will affect which vertex set.

Lemma 7. *After inserting a k-joint edge set $E_k = \{e_1, e_2, \ldots, e_p\}$ from a graph G, if we consider the vertex $v \in V(G) \setminus V_c$, only if $core(v) = k$ will increase its core number by at most 1.*

The proof of Lemma 7 can be found in Appendix C, available in the online supplemental material.

Similar as the insertion case, we can get the following result for the deletion case.

Lemma 8. *After deleting a k-joint edge set $E_k = \{e_1, e_2, \ldots, e_p\}$ from a graph G, if we consider the vertex $v \in V(G) \setminus V_c$, only if $core(v) = k$ will decrease its core number by at most 1.*

Now, we can further extend the conclusion to the case of inserting a joint edge set based on Lemma 7.

Lemma 9. *Given a joint edge set $\epsilon = E_{k_1} \cup E_{k_2} \cup \cdots \cup E_{k_q}$, where E_{k_i} $(1 \leq i \leq q)$ is a k_i-joint edge set and $k_i < k_j$ if $i < j$. After inserting ϵ into the graph G, each vertex $v \notin V_c$ will increase its core number by at most 1.*

Proof. It can be seen that inserting edges of ϵ into G all together has the same result with inserting E_{k_i} one by one. We next assume E_{k_i} is inserted one by one. To prove the lemma, we need to prove that if inserting E_{k_i} makes a vertex increase its core number from k_i to $k_i + 1$, its core number will not change any more when inserting E_{k_j} for $j > i$. Clearly, we only need to prove the above result for $E_{k_{i+1}}$. There are two cases we need to consider.

If $k_{i+1} > k_i + 1$, by Lemma 7, the core number of u will not increase any more when inserting $E_{k_{i+1}}$, since only vertices with core numbers of k_{i+1} may increase their core numbers.

We next consider the case of $k_{i+1} = k_i + 1$. We claim that if there is a vertex increasing its core number from k_i to $k_i + 2$ after the insertions of E_{k_i} and $E_{k_{i+1}}$, the vertex must have a neighbor which increases the core number from k_i to $k_i + 2$ as well during the insertions. Let u be a vertex whose core number is increased from k_i to $k_i + 2$ after inserting E_{k_i} and $E_{k_{i+1}}$. Notice that u does not connect to edges in $E_{k_{i+1}}$. Hence, the degree of u does not change when inserting $E_{k_{i+1}}$. Furthermore, if u increases its core number from $k_i + 1$ to $k_i + 2$ after inserting $E_{k_{i+1}}$ and its number of neighbors can increase by at most 1, it has at least $k_i + 2$ neighbors whose core numbers are no less than $k_i + 1$ before inserting $E_{k_{i+1}}$ and some of these neighbors have a core number of $k_i + 1$. The vertices in $N(u)$ whose core numbers are $k_i + 1$ before inserting $E_{k_{i+1}}$ are denoted by $P_{k_i+1}(u)$. It can be obtained that there must exist a vertex $w \in P_{k_i+1}(u)$ whose core number is k_i before inserting E_{k_i}, as otherwise, the core number of u is $k_i + 1$ before inserting E_{k_i}, which contradicts with our assumption.

Let V_2 denote the vertex set that each vertex in V_2 changes its core number from k_i to $k_i + 2$ after the insertions of E_{k_i} and $E_{k_{i+1}}$. Because inserting $E_{k_{i+1}}$ does not change the degrees of vertices in V_2, there must be a vertex $w \in V_2$ whose core number change is caused by the core number change of vertices in $N(w) \setminus V_2$, as otherwise no vertex in V_2 can change the core number. Assume w has at most k_i neighbors whose core numbers are no less than $k_i + 1$ in $N(w) \setminus V_2$ before inserting E_{k_i} and $E_{k_{i+1}}$, then inserting E_{k_i}, w can add at most 1 new neighbor whose core number is not smaller than k_{i+1}. When inserting $E_{k_{i+1}}$, w does not add new neighbors, so it has at most $k_i + 1$ neighbors whose core numbers are not smaller than $k_i + 1$. It cannot cause w increase its core number from $k_i + 1$ to $k_i + 2$. So w has at least $k_i + 1$ neighbors whose core numbers are not smaller than $k_i + 1$ in $N(w) \setminus V_2$ before inserting E_{k_i} and $E_{k_{i+1}}$. It concludes that $core(w) = k_i + 1$ before inserting E_{k_i} and $E_{k_{i+1}}$. However, it contradicts with the fact that $core(w)$ is k_i before insertions. The contradiction shows if a vertex changes its core number when inserting E_{k_i}, its core number will not change any more when inserting $E_{k_{i+1}}$. □

We have the following similar result for the deletion case.

Lemma 10. *Given a joint edge set $\epsilon = E_{k_1} \cup E_{k_2} \cup \cdots \cup E_{k_q}$, where E_{k_i} $(1 \leq i \leq q)$ is a k_i-joint edge set and $k_i < k_j$ if $i < j$. After deleting ϵ from the graph G, each vertex $v \notin V_c$ will decrease its core number by at most 1.*

In the above, we have shown that each vertex $v \in V \setminus V_c$ will change its core number by at most 1 with the insertion/deletion of a joint edge set. As for each vertex u in V_c, we can compute pre-core(u) using the core number of its neighbors. With the insertion/deletion of a joint edge set, its core number can also change by at most 1 from pre-core(u) by Lemma 1, as all its neighbors' core numbers can increase by at most 1. Thus, we can process all these edges in one iteration. Hence, we finally get the results for the insertion/deletion of a joint edge set.

Lemma 11. *Given a joint edge set ϵ, with the insertion/deletion of ϵ in the graph G, each vertex in $V(G)$ will change its core number by at most 1.*

4.3 Parallel Approach for Identifying the Core Number Changed Vertex Set After Deletion/Insertion of a JES

In [23], Sariyüce *et al.* give *TRAVERSAL* algorithms to identify the vertex set whose core number will be affected after the insertion/deletion of an edge. When multiple edges are updated, we need to generalize the result to the scenario of inserting/deleting a *JES*. We first define the K-Path-Tree of vertex that is used for *TRAVERSAL*.

Definition 15 (K-Path-Tree of a vertex u). *Given a vertex u such that $core(u) = K$, the* K-Path-Tree *of the vertex u in the graph G is a subgraph of G that is reachable from u and each vertex's core number equals to K in the subgraph. We denote the K-Path-Tree of vertex u as KPT_u for simplicity.*

It has been proved in [23] that with the insertion/deletion of an edge $\langle u, v \rangle$ and $core(u) \leq core(v)$, only for each vertex w

in KPT_u satisfying $SD(w) > core(u)$ (for insertion) or $SD(w) < core(u)$ (for deletion) will change its core number by 1. So when combining the idea behind the *TRAVERSAL* algorithm, we only need to find all available root vertices when inserting/deleting a *JES*.

For the insertion case, the root vertex set is defined as follows.

Definition 16 (Insertion Root Vertex Set). *Given a graph G, after inserting $JES = E_c \cup E_s$, the new graph is G'. The insertion root vertex set R_I contains the following vertices:*

(i) for each edge $\langle u, v\rangle \in E_s$ and $core(u) \leq core(v)$, $u \in R_I$;
(ii) for each vertex $v \in V_c$, we denote init-core(v) and pre-core(v) as its original core number in G and pre-core number in G', respectively. For each edge $\langle v, u\rangle$, if init-core(v) $\leq$ core(u) $\leq$ pre-core(v), $u \in R_I$.

The next Lemma 12 points out that R_I contains all available root vertices when inserting a *JES*.

Lemma 12. *After inserting a JES to a graph G, if we do not consider the vertex in V_c, only the vertices in $\bigcup_{u \in R_I} KPT_u$ have chances to increase the core numbers by at most 1.*

Proof. We define $P = \bigcup_{u \in R_I} KPT_u$. Assume a vertex $v \notin V_c$ and $v \notin P$, its core number increases by 1 from k. First, v does not have any neighbor in V_c whose pre-core number is no less than k, otherwise v will be added to R_I according to the definition of R_I. Thus, the increase of v's core number is not related to its neighbors in V_c as their final core numbers will be less than $k+1$. For other neighbors those are not in V_c and the core numbers are less than k, those core numbers won't help v increase to $k+1$. The reason is these neighbors' final core numbers change by at most 1 according to Lemma 9 and will be less than $k+1$. Besides, v does not insert any superior edge as otherwise it will also be added to R_I. Thus the reason for the increase of v's core number is because the core numbers of its neighbors increase from k to $k+1$ and all these neighbors are not contained in V_c and P. The reason for the increase of the core numbers of these neighbors is the same as v. As the original graph has a limited size, it cannot go on indefinitely. Thus our assumption is incorrect, that is to say, only vertices in P have chances to increase the core numbers by at most 1. □

For the deletion case, the root vertex set is defined as follows.

Definition 17 (Deletion Root Vertex Set). *Given a graph G, after deleting $JES = E_c \cup E_s$, the new graph is G'. The deletion root vertex set R_D contains the following vertices:*

(i) for each edge $\langle u, v\rangle \in E_s$ and $core(u) \leq core(v)$, $u \in R_D$;
(ii) for each vertex $v \in V_c$, we denote init-core(v) and pre-core(v) as its original core number in G and pre-core number in G', respectively. For each edge $\langle v, u\rangle$, if pre-core(v) $\leq$ core(u) $\leq$ init-core(v) , $u \in R_D$.

Similar to the insertion case, we can get the following Lemma 13.

Lemma 13. *After deleting a JES to a graph G, if we do not consider the vertex in V_c, only the vertices in $\bigcup_{u \in R_D} KPT_u$ have chances to decrease the core numbers by at most 1.*

After we have found all root vertices after inserting/deleting a JES, we traverse the K-Path-Tree of each vertex in root vertices to identify the change of core numbers. For distinct k, the traversing vertices are disjoint. Thus, we can split the tasks to different processes to take advantage of multi-core processors. As our algorithms spend most of the time to traverse, this parallelized task partition can improve the efficiency of our algorithms.

5 Incremental Core Maintenance

We propose the parallel algorithm to maintain each vertex's core number with the insertion of arbitrary edges E_I into the graph G in this section.

Algorithm 1. JointInsert(G, E_I, $core$)

Input
$G = (V, E)$ is the original graph;
E_I are edges to be inserted;
$core$ is the set of each vertex's core number before the insertion;
Output
Each vertex's updated core number;
Initially
$\mathcal{C} \leftarrow$ empty set;
▷ E_U is a mapping from core number k to the k-joint edge set
1 **while** E_I *is not empty* **do**
2 $E_U, V_c \leftarrow$ ComputeInsertEdgeSet(G, E_I, $core$);
3 $\mathcal{C} \leftarrow$ all core numbers of vertices in V_c;
▷$E_U[k]$ is a k-joint edge set
4 insert $\bigcup_{k \in \mathcal{C}} E_U[k]$ into G;
5 delete $\bigcup_{k \in \mathcal{C}} E_U[k]$ from E_I;
6 **for** *each core number $k \in \mathcal{C}$ in parallel* **do**
7 $V_k \leftarrow$ k-JointInsert(G, k, $E_U[k]$, $core$);
8 **for** *each $v \in V_k$* **do**
9 $core(v) \leftarrow core(v) + 1$;
10 **for** *each v in V_c* **do**
11 $core(v) \leftarrow$ re-compute using Equation (1);

Algorithm. The detailed algorithm to maintain each vertex's core number with the insertion of edges given in Algorithm 1 is executed until all edges in E_I have been processed(Line 1). In each iteration, the algorithm invokes the subroutine *ComputeInsertEdgeSet* to find a *JES* and divides it into disjoint k-joint edge sets (Line 2); then, it executes *k-JointInsert* algorithm (Lines 6-9) in parallel to find the vertex set that each vertex in the set changes its core number from k to $k+1$; finally, for each vertex in the central vertex set, its core number is recalculated using its neighbors' updated core numbers (Lines 10-11).

The Algorithm 2 finds a *JES* from unprocessed edges in E_I. Basically, this is done in two steps: (1) a central vertex set V_c is found from the superior vertex set (Line 1) and edges connected to these vertices are added to *JES*. Specially, for each vertex u in V_c, it updates its core number as pre-core value and saves its previous core number (Lines 2-4). At Lines 5-8, we record u's neighbors whose core numbers lie between init-core(u) and pre-core(u) as insertion root vertices according to Definition 16; (2) for each vertex $v \notin V_c$, we ensure that vertex v connects to at most one superior edge (Lines 9-15).

Algorithm 2. ComputeInsertEdgeSet($G, E_I, core$)

Input
$G = (V, E)$ is the graph;
E_I are edges to be inserted;
$core$ is the set of each vertex's core number before the insertion;
Output
A mapping from k to the k-joint edge set and a central vertex set;
Initially
$E_U \leftarrow$ empty map, $V_c \leftarrow \emptyset$;
$\forall v \in V$, $mark[v] \leftarrow$false;
1 $V_c \leftarrow$ a maximal 3$^+$-hop independent set of the superior vertex set after inserting E_I;
2 **for** *each vertex* v *in* V_c *in parallel* **do**
3 $init\text{-}core(v) \leftarrow core(v)$;
4 $core(v) \leftarrow pre\text{-}core(v)$;
5 **for** *each vertex* $u \in N_G(v)$ **do**
6 **if** $init\text{-}core(v) \leq core(u) \leq core(v)$ **then**
7 add $\langle u, v\rangle$ to $E_U[core(u)]$;
8 $mark[u] \leftarrow$true;
9 **for** *each edge* $\langle x, y\rangle \in E_I$ *with* $core(y) \leq core(x)$ *in parallel* **do**
10 **if** $y \in V_c$ *or* ($core(y) < core(x)$ *and* $mark[y]$ = *false*) **then**
11 add $\langle x, y\rangle$ to $E_U[core(y)]$;
12 $mark[y] \leftarrow$true;
13 **if** $core(x) = core(y)$ *and* $mark[x] = mark[y]$ = *false* **then**
14: $mark[x] \leftarrow$ true; $mark[y] \leftarrow$ true;
15: add $\langle x, y\rangle$ to $E_U[core(x)]$;
16: **return** E_U, V_c;

After the *JES* is found, for each k-joint edge set, we use Algorithm 3 to find out all vertices whose core numbers will change. For each root vertex u, we first compute the SD value of each vertex in the KPT_u and push it to a stack (Lines 1-4). Then it executes until the stack is empty (Line 5). In the loop, starting from each root vertex u, it does a depth-first search (DFS) procedure to record all vertices in KPT_u whose SD values are greater than k (Lines 6-10). When we encounter with a vertex w that $SD(w) \leq k$ during the DFS procedure, another DFS process rooted from vertex w (we call this DFS as a negative DFS) will be performed to eliminate vertices those cannot find $k+1$ neighbors whose SD values are not less than $k+1$ (Lines 12-20). Finally, the remaining vertices will change their core numbers from k to $k+1$ (Lines 21-23).

Performance Analysis. To analyze the efficiency of our incremental algorithm, we first introduce some notations to measure the time complexity of our algorithm.

Let $G' = (V, E \cup E_I)$. For a vertex $v \in V$, let $N_2(v)$ denote the set of 2-hop neighbors of v, i.e., $N_2(v) = \{N_{G'}(v) \cup \bigcup_{u \in N_{G'}(v)} N_{G'}(u)\}$, and $N_2[v] = \{v\} \cup N_2(v)$. We denote V_S as the superior vertex set, and Δ_I as $\max\{|N_2[v] \cap V_S| : v \in V_S\}$, which is the maximum number of different central vertex sets that can be selected in the algorithm.

We denote $G_i = (V_i, E_i)$ as the new graph after the ith iteration. We denote S as the *JES* to be inserted. Let $G_S = (V_{i-1}, E_{i-1} \setminus S)$, $K(G_S)$ is the set of core numbers of vertices in $V(G_S)$. For a given $k \in K(G_S)$, let $V_S(k)$ be the vertex set that the core number of each vertex in the set equals to k and $E_S(k)$ is the set of edges connected to vertices in $V_S(k)$. For simplicity, we define $n_{S,k} = |V_S(k)|$ and $m_{S,k} = |E_S(k)|$, respectively.

Algorithm 3. k-JointInsert($G, k, E_k, core$)

Input
$G = (V, E)$ is the current new graph;
E_k is the k-joint edge set;
$core$ is the set of each vertex's core number before the insertion;
Output
A vertex set that each vertex in the set will change its core number;
Initially
$Stk \leftarrow$ empty stack, $V_t \leftarrow$ empty set;
$\forall v \in V$, evicted[v] $\leftarrow$false, visited[v] $\leftarrow$false, cd[v] $\leftarrow$0;
$\triangleright$ cd[v] counts the number of v's neighbors whose core numbers are no less than k
1 **for** *each edge* $\langle u, v\rangle \in E_k$ **do**
2 **for** *each vertex* $w \in KPT_u$ **do**
3 $cd[w] \leftarrow SD$ value of w;
4 Stk.push(u);$visited[u] \leftarrow$ true;
5 **while** *not Stk.empty()* **do**
6 $u \leftarrow Stk$.pop();
7 **if** $cd[u] > k$ **then**
8 **for** *each edge* $\langle u, v\rangle \in E$ **do**
9 **if** *not visited*[v] *and* $core(v) = k$ *and* $cd[v] > k$ **then**
10 Stk.push(v);$visited[v] \leftarrow$ true;
11 **else**
12 **if** *not evicted*[u] **then**
13 $S_2 \leftarrow$ empty stack;
14 S_2.push(u);$evicted[u] \leftarrow$ true;
15 **while** *not* S_2 *empty* **do**
16 $x \leftarrow S_2$.pop();
17: **for** *each* $\langle x, y\rangle \in E$ *such that* $core(y) = k$ **do**
18 $cd[y] \leftarrow cd[y]$ - 1;
19 **if** *not evicted*[y] *and* $cd[y]$=k **then**
20 S_2.push(y); $evicted[y] \leftarrow$ true;
21 **for** *each vertex* v *in* V **do**
22 **if** *not evicted*[v] *and visited*[v] **then**
23 $V_t \leftarrow V_t \cup \{v\}$;
24 **return** V_t;

In Algorithm 2, the time complexity is dominated by the for loop in lines 5-8 which is $O(\Delta(G))$. In Algorithm 3, the complexity of the algorithm consists of two parts: compute SD value of each vertex in $V_S(k)$ and traverse vertices in $V_S(k)$. The complexity of the former part is $O(m_{S,k})$ since we need to collect each vertex's neighbors' core numbers to compute its SD value. For the latter part, when we traverse the vertices in G_S, the DFS procedure visits each vertex in $V_S(k)$ at most once. And $L_{S,k} = max_{u \in V_{S(k)}}\{SD(u) - core_{G_S}(u), 0\}$ is the maximum number of times a vertex will be accessed during the negative DFS procedure, as each vertex v is accessed, $cd(v)$ will be decreased by 1 (Line 18 in Algorithm 3). By adding the two parts, we have the time complexity of Algorithm 3 is $C_S = max\{m_{S,k} + L_{S,k} \times n_{S,k}, k \in K(G_S)\}$.

Combining all the above analysis, we can get the following result.

Theorem 14. *The time complexity of Algorithm 1 updating the core numbers of vertices is* $O(\Delta_I \times max\{C_S, S \subseteq E_I\})$.

Proof. Consider a vertex v, in each iteration, if there are vertices in $N_2[v]$, one of these vertices must be selected into the

central vertex set. Hence, $O(\Delta_I)$ iteration times are needed to process all vertices in the superior vertex set. After that, there is at most one another iteration to process all remaining superior edges. The total number of iterations is $\Delta_I + 1$. Since the time complexity of each iteration is dominated by the time complexity of Algorithm 3, we can bound the time complexity of the incremental algorithm as stated. □

6 Decremental Core Maintenance

We propose the parallel algorithm to maintain each vertex's core number with the deletion of arbitrary edges E_D from the graph G in this section.

Algorithm. The detailed algorithm to maintain each vertex's core number with the deletion of edges given in Algorithm 4 is executed until all edges in E_D have been processed (Line 1). Similar to the incremental algorithm, in each iteration, it invokes the subroutine *ComputeDeleteEdgeSet* to find a *JES* and divides it into disjoint k-joint edge sets (Line 2); then it executes k-JointDelete algorithm in parallel to find vertices whose core numbers change from k to $k-1$ (Lines 6-7); finally, for each vertex in the central vertex set, its core number is recalculated using its neighbors' updated core numbers (Lines 10-11).

Algorithm 5 finds a *JES* from unprocessed edges in E_D. The algorithm is similar to Algorithm 2 except that for each root vertex $u \in V_c$, we record u's neighbors whose core numbers between pre-core(u) and init-core(u) as the root vertex deletion according to Definition 17 at Line 6. When we traverse vertices in Algorithm 6, the difference is that we evict vertices whose cd values are less than k (Lines 6,8) in the DFS traversing and in the negative DFS traversing (Line 17). The reason is these vertices have less than k neighbors whose core numbers are not less than k.

Algorithm 4. JointDelete(G, E_D, $core$)

Input
$G = (V, E)$ is the original graph;
E_D are edges to be deleted;
$core$ is the set of each vertex's core number before the deletion;
Output
Each vertex's updated core numbers;
Initially
$\mathcal{C} \leftarrow$ empty core set;
$\triangleright$ E_U is a mapping from the core number k to the k-joint edge set

```
1  while E_D is not empty do
2    E_U, V_c ← ComputeDeleteEdgeSet(G, E_D, core);
3    C ← all core numbers of vertices in E_U;
     ▷ E_U[k] is a k-joint edge set
4    delete ⋃_{k∈C} E_U[k] from G;
5    delete ⋃_{k∈C} E_U[k] from E_D;
6    for each core number k ∈ C in parallel do
7      V_k ←k-JointDelete(G, E_U[k], core);
8      for each v in V_k do
9        core(v) ← core(v) − 1;
10   for each v in V_c do
11     core(v) ← re-compute using Equation (1);
```

Performance Analysis. To analyze the efficiency of our decremental algorithm, we first define some notations to measure the time complexity of our algorithm.

For graph $G = (V, E)$, E_D are edges to be deleted and V_D is the set of vertices associated with E_D. We denote V_S as the superior vertex set of V_D. Let $G' = (V, E \setminus E_D)$. For a vertex $v \in V$, $N_2(v) = \{N_{G'}(v) \cup \bigcup_{u \in N_{G'}(v)} N_{G'}(u)\}$. We denote Δ_D as $\max\{|N_2(v) \cap V_S|, v \in V_S\}$. It is the maximum number of different central vertex sets that can be selected in the algorithm.

Algorithm 5. ComputeDeleteEdgeSet(G, E_D, $core$)

Input
$G = (V, E)$ is the graph;
E_D are edges to be deleted;
$core$ is the set of each vertex's core number before the deletion;
Output
A mapping from k to the k-joint edge set and the central vertex set;
Initially
$E_U \leftarrow$ empty map, $V_c \leftarrow \emptyset$;
$\forall v \in V$, $mark[v] \leftarrow$ false;

```
1  V_c ← a maximal 3+-hop independent set of a superior vertex
     set after deleting E_D;
2  for each v in V_c in parallel do
3    init-core(v) ← core(v);
4    core(v) ← pre-core(v);
5    for each u in N_G(v) do
6      if core(v) ≤ core(u) ≤ init-core(v) then
7        add ⟨u, v⟩ to E_U[core(u)];
8        mark[u] ← true;
9  for each edge ⟨x, y⟩ ∈ E_D with core(y) ≤ core(x) in parallel do
10   if y ∈ V_c or
     (core(y) < core(x) and mark[y]=false) then
11       add ⟨x, y⟩ to E_U[core(y)];
12       mark[y] ← true;
13   if core(x) = core(y) and mark[x]=mark[y]=false then
14     mark[x] ← true; mark[y] ← true;
15     add ⟨x, y⟩ to E_x[core(x)];
16 return E_U, V_c;
```

We denote $G_i = (V_i, E_i)$ as the new graph after the ith iteration. In each iteration, S is the *JES* selected to be deleted, $G_S = (V_{i-1}, E_{i-1} \setminus S)$. We denote $K(G_S)$ as the set of core numbers of vertices in $V(G_S)$. For $k \in K(G_S)$, let $V_S(k)$ be the vertex set that each vertex's core number equals to k and $E_S(k)$ is the set of edges connected to vertices in $V_S(k)$. For simplicity, we define $n_{S,k} = |V_{S,k}|$ and $m_{S,k} = |E_S(k)|$, respectively. We also define $L_{S,k} = max_{u \in V_{S(k)}}\{SD(u) - core_{G_S}(u), 0\}$. Similar as the performance analysis discussed in the incremental algorithm, the total time complexity of iteration i is $C_S = max\{m_{S,k} + L_{S,k} \times n_{S,k}, k \in K(G_S)\}$.

Using a similar analysis as that of Theorem 14, we can conclude the following result.

Theorem 15. *The time complexity of Algorithm 4 updating the core numbers of vertices is* $O(\Delta_D \times max\{C_S, S \subseteq E_D\})$.

7 Experimental Studies

We have conducted experimental studies using 12 real-world graphs, 3 synthetic data sets and 3 temporal networks. We first report the performance, scalability and stability of our algorithms. Then, we show the parallelism of our algorithms. Finally, we compare our algorithms with the single edge

TABLE 2
Attributes of Real-World Graphs

Dataset	n=$\|V\|$	m=$\|E\|$	avg. deg	max k
GW(Gowalla)	0.19M	0.91M	9.67	51
DB(DBLP)	0.30M	1.00M	6.62	113
CA(RoadNet-CA)	1.97M	2.77M	2.82	3
YT(Youtube)	1.08M	2.85M	5.27	51
BS(BerkStan)	0.65M	6.34M	19.41	201
PT(Patents)	3.60M	15.75M	8.75	64
PC(coPapersCiteseer)	0.43M	16.04M	73.72	384
PK(Pokec)	1.56M	21.27M	27.32	47
LJ(LiveJournal)	3.81M	33.07M	17.35	360
OK(Orkut)	2.93M	111.76M	76.28	253
UK(uk-2002)	18.52M	298.11M	16.10	943
AC(arabic-2005)	22.74M	639.99M	28.14	3247

processing traversal algorithms proposed in [23] and [18], the matching based algorithms proposed in [14] and the superior edge based algorithms proposed in [25]. We timed the while loop in Algorithms 1 and 4, that is the time cost by the complete algorithm. We implement all algorithms in C++ language and compile the source code using g++ with -O3 optimization level. The hardware environment is 64-bit Linux machine with 60 vCPUs and 128GB size memory.

Algorithm 6. k-JointDelete($G, k, E_k, core$)

Input
G is the current new graph;
E_k is the k-joint edge set, E_k;
$core$ is the set of each vertex's core number before the deletion;
Output
A vertex set that each vertex in the set will change its core number;
Initially
$V_t \leftarrow$ empty set;
$\forall v \in V$, evicted$[v] \leftarrow$false, cd$[v] \leftarrow$0;
$\triangleright$ cd$[v]$ counts the number of v's neighbors whose core numbers are no less than k

```
1  for each ⟨u, v⟩ ∈ E_k do
2    for each vertex w ∈ KPT_u do
3      cd[w] ← SD value of w;
4  for each ⟨u, v⟩ ∈ E_k do
5    C ← empty set;
6    if not evicted[u] and cd[u] < k then
7      add u to C;
8    if core(u) = core(v) and not evicted[v] and cd[v] < k then
9      add v to C;
10   for each vertex w ∈ C do
11     Stk ← empty stack;
12     Stk.push(w); evicted[w] ←true;
13     while not Stk empty do
14       x ← S.pop();
15       for each ⟨x, y⟩ ∈ E that core(y) = k do
16         cd[y] ← cd[y]-1;
17         if cd[y] < k and not evicted[y] then
18           Stk.push(y); evicted[y] ←true;
19 for each vertex v in V do
20   if evicted[v] then
21     V_t ← V_t ∪ {v};
22 return V_t;
```

TABLE 3
Attributes of Temporal Networks

Dataset	n=$\|V\|$	Temporal Edges	Static Edges
SU(Super User)	0.197M	1.44M	0.925M
WK(Wiki-talk)	1.14M	7.8M	3.3M
ST(Stack Overflow)	2.6M	63.5M	36.2M

Datasets. The 12 real-world graphs can be downloaded from Stanford Network Analysis Platform [15] and 10th DIMACS Implementation Challenge [22] and [7], including Web Graphs (BerkStan), Social Networks (LiveJournal and Pokec), Ground-truth Community Networks (DBLP, Youtube and Orkut), Citation Networks (Patents), Road Network of California (RoadNet-CA), Location-Based Social Networks (Gowalla), coPapersCiteseer [22], uk-2002 [7], arabic-2005 [7]. We have converted directed graphs to undirected ones in order to adapt to our algorithms. The statistics of real-world graphs are listed in Table 2. In this table, "max k" means the maximum core number of the graph.

We generate the synthetic graphs using Stanford Network Analysis Platform system with the following three models: the Barabasi-Albert (BA) preferential attachment model [5] that the degree of each vertex is k; the R-MAT (RM) graph model [9] that generates graph structures similar to real-world graphs; the Block Two-Level Erdös-Rényi (BTER) graph model [13] that accurately captures the observable properties of many real world social networks. In our experiments, we fix the average degree of every synthetic graph as 8. Specifically, for the BA graph, each vertex will have the same core number 8 under the average degree assumption. This special core number distribution of the BA graph will be used as the extreme case for testing the scalability of our algorithms. The reason is that the core number distribution covers a wide range in real-world graphs [14]. The details of three temporal networks WK, SU and ST are shown in Table 3.

7.1 Stability Evaluation

To test the stability of our algorithms, we select 8 real-world graphs. In each graph, we randomly select P_i edges where $P_i = 10^i, 1 \le i \le 5$. We first remove these edges from the original graph and then insert back to this graph. The processing time cost by each edge for insertion and deletion cases are shown Figs. 3a and 3b, respectively. It can be shown that as the exponential growth of the updated edges, the time cost by each edge has a downward trend. It is because if there are more updated edges, we can process more edges in each

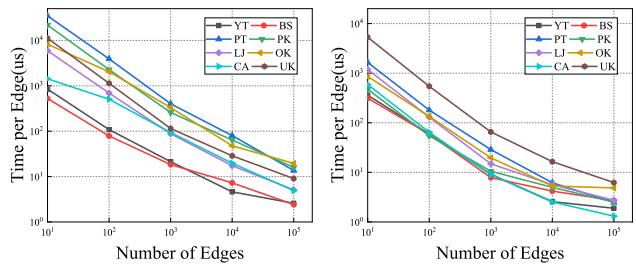
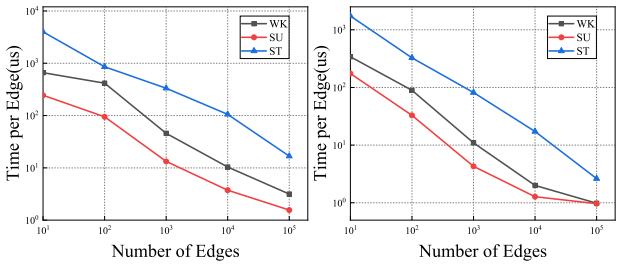


Fig. 3. Influence of the number of updated edges in real-world graphs.



(a) Insertion (b) Deletion

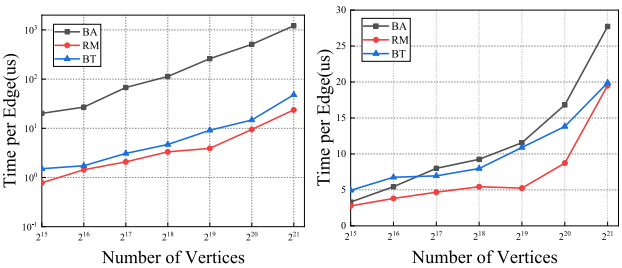
Fig. 4. Influence of the number of updated edges in temporal networks.

iteration and avoid unnecessary access. Consider the situation in Fig. 2a, if we insert edge set $\{< v6, v1 >, < v6, v3 >, < v6, v4 >, < v6, v5 >\}$ one by one using TRAVERSAL algorithm, we need to visit vertex $v1$ four times, vertex $v3$ three times, vertex $v4$ twice, vertex $v5$ once, respectively. However, our proposed algorithm can process these edges using one iteration and visit vertices $v1, v3, v4, v5$ once. Thus in the real-world graphs, as we delete/insert more edges, the time cost by each edge decreases.

Next, we use temporal graphs to test the performance of our algorithms. In each temporal graph, we select five time points $T_i, 1 \leq i \leq 5$ that the number of edges after T_i is around 10^{6-i}. The processing time cost by each edge for the insertion and deletion cases are shown in Figs. 4a and 4b, respectively. We can also see that the average processing time cost by each edge decreases as the number of inserted/deleted edges increases.

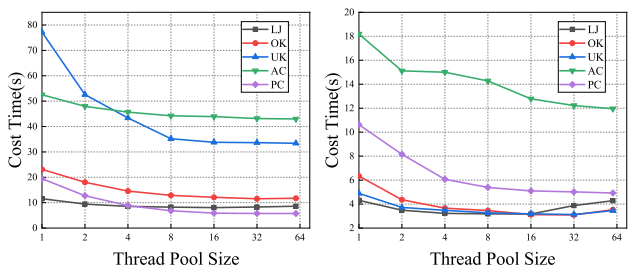
7.2 Scalability Evaluation

To test the scalability of our algorithms, we use 3 synthetic graphs. In each graph, the average degree is fixed as 8 and the number of vertices varies from 2^{15} to 2^{21}. We randomly select $10K$ updated edges, Figs. 5a and 5b show the result. We can conclude that as the size of the original graph grows exponentially, the average time cost by each edge has a gentle growth trend. The experimental results mean that our algorithms are suitable for processing large-scale graphs and can achieve good scalability. However, it takes more processing time for BA graphs as each vertex's core number is 8. Our algorithms are not suitable to handle this situation for two reasons. First, the size of the traversed vertex set grows exponentially as the graph size grows. Second, only one thread is used to execute the algorithm as it has only one 8-joint edge set. However, in real-worlds graphs, as the core number distribution covers a wide range [14], the extreme case will not occur.



(a) Insertion (b) Deletion

Fig. 5. Influence of original graph's size.



(a) Insertion (b) Deletion

Fig. 6. Influence of the thread pool size.

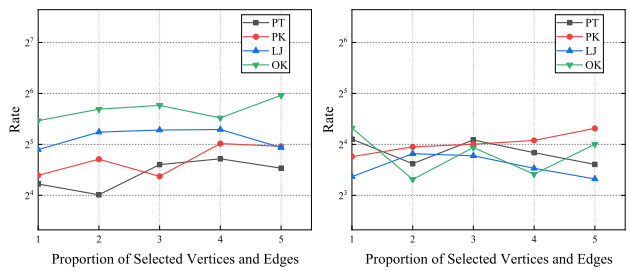
7.3 Parallelism Evaluation

To evaluate the parallelism of our algorithms, we vary the number of parallel threads to execute our algorithms. We implement a thread pool using Linux POSIX threads to submit tasks. In the implementation, we divide the edges of a joint edge set into different groups and define each group as a k-joint edge set. Then we process all k-joint edge sets in parallel. Each thread processes a specific k-joint edge set for load balancing.

We experiment on 5 real-world graphs LJ, OK, UK, AC and PC. For each graph, the number of updated edges are kept as 1 $million$ and the size of the thread pool ranges from 1 to 60. We show the results in Figs. 6a and 6b. It can be seen that as the size of the thread pool grows, the total processing time is about half of the time cost by single-core processing. The main reason for limiting parallelism is the core numbers are not well-distributed, so some threads need to traverse a large subgraph that takes a great amount of time. In addition, when the maximum core number of a graph is smaller than the number of threads, some threads have to be wasted which results in extra overheads. As shown in Fig. 6b, the cost time has slightly increased when the number of threads exceeds a certain value. In general, when the size of the processing graph is large and updates lots of edges, the parallelism of our algorithms can save considerable time.

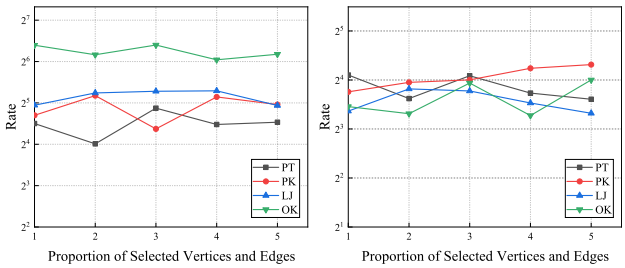
7.4 Comparisons With Existing Algorithms

The comparison between our JES based algorithms(JBA) with existing matching based algorithms(MBA) in [14], superior edge based algorithms(SBA) in [25], TRAVERSAL algorithms in [23] and the algorithms in [18] are made in this section. When compared with MBA and SBA, we use 4 real-world graphs PT, PK, LJ, OK and randomly select i‰vertices and i‰edges where $1 \leq i \leq 5$ as updated sets. The numbers of threads SBA, MBA and JBA used are the same and the value is 16. The comparison results between JBA and MBA are shown in Figs. 7a and 7b. The rate (y-axis) means the ratio of the time cost by existing



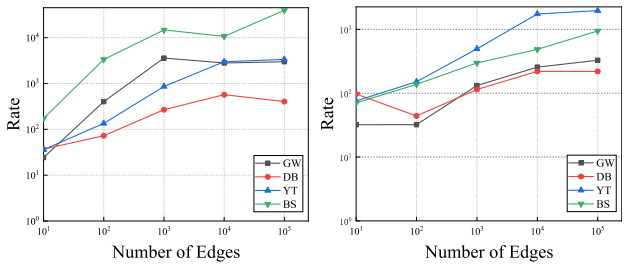
(a) Insertion (b) Deletion

Fig. 7. Comparison with the Algorithm MBA.



(a) Insertion (b) Deletion

Fig. 8. Comparison with the Algorithm SBA.



(a) Insertion (b) Deletion

Fig. 10. Comparison with the TRAVERSAL Algorithm.

algorithms over the time cost by our algorithms. For the insertion case, the speedup can achieve at most 60 in the largest graph OK. This is because in larger graphs, our algorithms can select more central vertices in each iteration and can process more edges. For the deletion case, the speedup is between 10 and 20. The results of the comparisons between JBA and SBA are shown in Figs. 8a and 8b. We can get the similar speedup with the MBA. One of the most important reasons why our algorithms beat the other two algorithms is that our algorithms can process vertices updating efficiently. Figs. 9a and 9b show the iteration times used for three algorithms. It can be seen that the iteration times cost by MBA and SBA are far more than the iteration times needed by our algorithms. Especially for the largest graph OK, the other two algorithms need over 1,000 iterations while our algorithms need only about 35 iterations.

The comparison results compared with the TRAVERSAL algorithms conducted on 4 real-world graphs GW, DB, YT, BS are shown in Figs. 10a and 10b. The numbers of threads our algorithms and TRAVERSAL algorithms used are 16 and 1 respectively as TRAVERSAL algorithms are sequential. We randomly select $\{10, 100, 1K, 10K, 100K\}$ edges as updated sets in each graph. The results indicate that when multiple edges can be processed simultaneously, our algorithms outperform TRAVERSAL algorithms by up to four orders of magnitude. The speedup comes from two main reasons: (i) multiple edges processing in one iteration reduces the unnecessary visiting of vertices compared with processing these edges one by one; (ii) parallel processing of our algorithms.

We also compare with the single edge processing algorithms proposed in [18]. The number of threads our algorithms and the algorithms [18] used is 16 and 1 respectively as the latter one is sequential. The results are shown in Figs. 11a and 11b. It can also conclude that our algorithms outperform the existing single edge processing algorithms by almost three orders of magnitude.

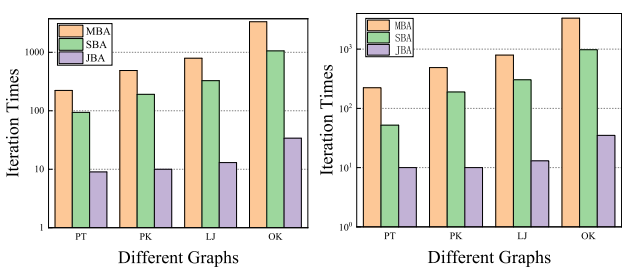
8 Application in Distributed Core Decomposition

In this section, we will show how to reduce the time complexity of an existing distributed k-core decomposition algorithm by using the "joint edge set" proposed in our core-maintenance algorithms.

In [20], Montresor *et al.* give a distributed algorithm to calculate each node's core number based on the property of locality of the k-core decomposition. The algorithm works as follows: each node produces an *estimate* of its own core number and communicates to its neighbors; at the same time, it receives estimates from its neighbors and uses them to recompute its own estimate; in the case of a change, a new value is sent to the neighbors and the process goes on until convergence. In the paper, the authors use the degree as each node's initial core number estimate as each node's core number is not greater than its degree. It is proved that the algorithm will eventually converge to the correct core number and the time complexity (the converging time) is bounded by $1 + \sum_{u \in V} [d(u) - k(u)]$ where $d(u)$ is the initial degree of u and $k(u)$ is the actual core number of u. In dynamic graphs, taking the insertion case as an example, the time complexity of the distributed core decomposition is $1 + \sum_{u \in V'} [d'(u) - k'(u)]$ where $d'(u)$ and $k'(u)$ are the degree and core number of u in the updated graph $G'(V', E')$, respectively. Since the time complexity is bounded by the sum of each node's degree, the time complexity will be greatly increased as more edges are inserted in the graph.

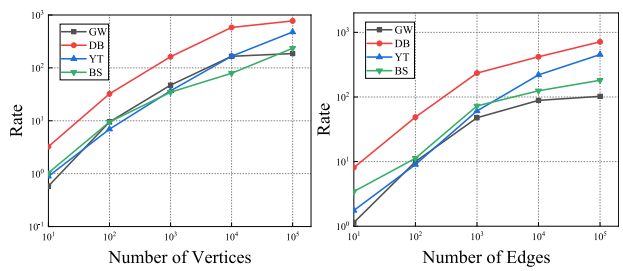
In our core maintenance algorithms, the structure of the "joint edge set" (JES) can help get a more accurate upper bound of the initial core number estimate than the degree, so that the convergence procedure is accelerated.

When inserting a JES, the core number of each vertex except the central vertices can increase by at most 1. From the proof of Theorem 14, we know the iteration times of our



(a) Insertion (b) Deletion

Fig. 9. Comparison of iteration times.



(a) Insertion (b) Deletion

Fig. 11. Comparison with the Algorithm in [18].

algorithm is at most $\Delta_I + 1$ in the insertion case. For each vertex which is not in the central vertex set, its core number can increase by at most $\Delta_I + 1$. For each vertex in the central vertex set, it can obtain its pre-core by communicating with its neighbors and then its core number will increase by at most $\Delta_I + 1$ from its pre-core number (c.f. Lemma 1). As already mentioned, for a given vertex, its core number is not greater than its degree. Thus, for a vertex u with core number k in the original graph, its core number in the updated graph is at most $c'(u) = min\{k + \Delta_I + 1, d'(u)\}$, where $d'(u)$ is the degree of u in the updated graph. Now the complexity of the distributed core decomposition algorithm is bounded by $1 + \sum_{u \in V} [c'(u) - k'(u)]$ where $k'(u)$ is the core number of u in the updated graph.

9 Conclusion

We propose new algorithms that can process multiple edges/vertices insertions/deletions in the dynamic graphs. Based on a structure of Joint Edge Set, we present faster parallel algorithms for both the incremental and decremental core maintenance problems. Extensive experiments show the superiority of our approach comparing with previous single-edge processing algorithms, matching based algorithms and superior edge set based algorithms. Meanwhile, our algorithms exhibit good scalability and stability in practice.

In the future, we will futher optimize our algorithms and multi-core implementations in three aspects. One is to find a more efficient algorithm that can find out the vertex set in which each vertex changes its core number with the insertion/deletion of an edge from the graph. The second is trying to find a more efficient structure that can accommodate more edges in one iteration. For example, our protocol requires the central vertices are 3^+-hop independent in each iteration. It will be interesting to check if it is possible to relax this assumption. The third one is trying to find a more efficient load balancing approach to adapt to different core number distributions of various graphs so that the protocol can better harness the multi-core parallelism.

Acknowledgments

The first author would like to thank Xiaohui Zhang for helping to implement the experiment. This work was supported in part by the National Key Research and Development Program of China under Grant No. 2018YFB1003203 and National Natural Science Foundation of China Grants 61572216, 61832006, 61971269, 61672321, and 61832012.

References

[1] H. Aksu, M. Canim, Y. C. Chang, I. Korpeoglu, and Ö. UIusoy, "Distributed K-core view materialization and maintenance for large dynamic graphs," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 10, pp. 2439–2452, Oct. 2014.

[2] J. I. Alvarezhamelin, L. Dall'Asta, A. Barrat, and A. Vespignani, "Large scale networks fingerprinting and visualization using the K-core decomposition," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2006, pp. 41–50.

[3] 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.

[4] B. G. D. Bader and C. W. Hogue, "An automated method for finding molecular complexes in large protein interaction networks," *BMC Bioinf.*, vol. 4, 2003, Art. no. 2.

[5] A. L. Barabasi and R. Albert, "Emergence of scaling in random networks," *Sci.*, vol. 286, no. 5439, pp. 509–512, 1999.

[6] V. Batagelj and M. Zaversnik, "An O(m) algorithm for cores decomposition of networks," in *CoRR*, vol. cs.DS/0310049, 2003, *arXiv:cs/0310049*.

[7] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. 13th Int. World Wide Web Conf.*, 2004, pp. 595–601.

[8] S. Carmi, S. Havlin, S. Kirkpatrick, Y. Shavitt, and E. Shir, "A model of internet topology using K-shell decompositio," *Proc. Nat. Academy Sci. United States America*, vol. 104, no. 27, pp. 11150–11154, 2007.

[9] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proc. SIAM Int. Conf. Data Mining*, 2004, pp. 442–446.

[10] J. Cheng, Y. Ke, S. Chu, and M. T. Özsu, "Efficient core decomposition in massive networks," in *Proc. IEEE 27th Int. Conf. Data Eng.*, 2011, pp. 51–62.

[11] N. S. Dasari, R. Desh, and M. Zubair, "ParK: An efficient algorithm for K-core decomposition on multicore processors," in *Proc. IEEE Int. Conf. Big Data*, 2015, pp. 9–16.

[12] S. Du, G. Song, H. Hong, and D. Lua, "Learning dynamic dependency network structure with time lag," *Sci. China Inf. Sci.*, vol. 61, no. 5, pp. 052101:1–052101:16, 2018.

[13] P. Erdös and A. Renyi, "On the evolution of random graphs," in *Publications Math. Inst. Hungarian Acad. Sci.*, vol. 5, pp. 17–61, 1960.

[14] H. Jin, N. Wang, D. X. Yu, Q. S. Hua, X. H. 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.

[15] L. Jure and K. Andrej, "SNAP datasets: Stanford large network dataset collection," 2014. [Online]. Available: http://snap.stanford.edu/data

[16] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo, "K-core decomposition of large networks on a single PC," *Proc. VLDB Endowment*, vol. 8, no. 1, pp. 13–23, 2016.

[17] M. Kitsak *et al.*, "Identification of influential spreaders in complex networks," *Nature Phys.*, vol. 6, no. 11 pp. 888–893, 2010.

[18] R. H. Li, J. X. Yu, and R. Mao, "Efficient core maintenance in large dynamic graphs," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 10, pp. 2453–2465, Oct. 2014.

[19] P. Meyer, H. Siy, and S. Bhowmick, "Identifying important classes of large software systems through K-core decomposition," *Advances Complex Syst.*, vol. 17, no. 07n08, 2014, Art. no. 1550004.

[20] 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.

[21] S. Sallinen, R. Pearce, and M. Ripeanu, "Incremental graph processing for on-line analytics," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2019, pp. 1007–1018.

[22] P. Sanders and C. Schul, "10th DIMACS implementation challenge," 2012. [Online]. Available: http://www.cc.gatech.edu/dimacs10

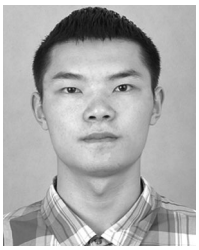
[23] A. E. Sariyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Catalyürek, "Incremental K-core decomposition: Algorithms and evaluation," *VLDB J.*, vol. 25, no. 3, pp. 425–447, 2016.

[24] A. E. Sariyuce, C. Seshadhri, A. Pinar, and U. V. Catalyurek, "Finding the hierarchy of dense subgraphs using nucleus decompositions," in *Proc. 24th Int. Conf. World Wide Web*, 2015, pp. 927–937.

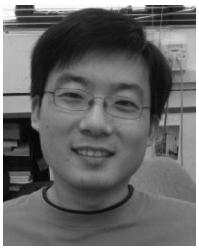
[25] N. Wang, D. X. Yu, H. Jin, C. Qian, X. Xie, and Q. S. Hua, "Parallel algorithm for core maintenance in dynamic graphs," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst.*, 2017, pp. 2366–2371.

[26] D. Wen, L. Qin, Y. Zhang, X. Lin, and J. X. Yu, "I/O efficient core graph decomposition at web scale," in *Proc. IEEE 32nd Int. Conf. Data Eng.*, 2016, pp. 133–144.

Qiang-Sheng Hua received the BEng and MEng degrees from the School of Information Science and Engineering, Central South University, Changsha, China, in 2001 and 2004, respectively, and the PhD degree from the Department of Computer Science, University of Hong Kong, Hong Kong, China, in 2009. He is currently an associate professor with the Huazhong University of Science and Technology, China. His research interests include parallel and distributed computing, including algorithms, and implementations in real systems. He is a member of the IEEE.



Yuliang Shi received the BE degree from Computer School, Hunan University, Changsha, China, in 2016. He is currently working toward the master's degree in the Department of Computer Science, Huazhong University of Science and Technology (HUST), Wuhan, China. His research interests include dynamic graph and parallel computing.

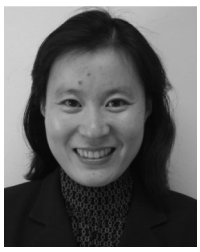


Dongxiao Yu received the BSc degree from the School of Mathematics, Shandong University, Jinan, China, in 2006 and the PhD degree from the Department of Computer Science, University of Hong Kong, Hong Kong, China, in 2014. He is currently a professor with the School of Computer Science and Technology, Shandong University. His research interests include wireless networks and distributed computing. He is a member of the IEEE.

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

Jiguo Yu received the PhD degree in operation research and control theory from Shandong University, Jinan, China, in 2004. He is currently a professor with the School of Computer Science and Technology, Qilu University of Technology. He is interested in designing and analyzing algorithms for computationally hard problems in communication networks and systems. He is a member of the ACM, and a senior member of the IEEE and China Computer Federation.

Zhipeng Cai received the PhD degree from the Department of Computing Science, University of Alberta, Edmonton, Canada. He is currently an associate professor with the Department of Computer Science, Georgia State University. His research areas focus on cyber-security, privacy, networking, and big data. He is a member of the IEEE.



Xiuzhen Cheng received the MS and PhD degrees in computer science from the University of Minnesota-Twin Cities, in 2000 and 2002, respectively. She is currently a professor with the Department of Computer Science, George Washington University, Washington, DC. Her current research interests include privacy-aware computing, wireless and mobile security, cyber physical systems, mobile computing, and algorithm design and analysis. She is a fellow of the IEEE.

Hanhua Chen received the PhD degree in computer science and engineering from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2010. He is currently a professor with the School of Computer Science and Technology, HUST, China. His research interests include distributed systems and BigData processing systems. He received the National Excellent Doctorial Dissertation Award of China, in 2012. He is a member of the IEEE.