Core Maintenance in Dynamic Graphs: A Parallel Approach Based on Matching

Hai Jin, *Senior Member, IEEE*, Na Wang, Dongxiao Yu, *Member, IEEE*, Qiang-Sheng Hua, *Member, IEEE*, Xuanhua Shi, *Senior Member, IEEE*, and Xia Xie

Abstract—The core number of vertices is a basic index depicting cohesiveness of a graph, and has been widely used in large-scale graph analytics. In this paper, we study the update of core numbers of vertices in dynamic graphs with edge insertions/deletions, which is known as the core maintenance problem. Different from previous approaches that just focus on the case of single-edge insertion/deletion and sequentially handle the edges when multiple edges are inserted/deleted, we investigate the parallelism in the core maintenance procedure. Specifically, we show that if the inserted/deleted edges constitute a matching, the core number update with respect to each inserted/deleted edge can be handled in parallel. Based on this key observation, we propose parallel algorithms for core maintenance in both cases of edge insertions and deletions. Extensive experiments are conducted to evaluate the efficiency, stability, parallelism and scalability of our algorithms on different types of real-world, synthetic graphs and temporal networks. Comparing with former approaches, our algorithms can improve the core maintenance efficiency significantly.

Index Terms—Graph analysis, dynamic graph, core number maintenance, parallel algorithm

1 Introduction

As a basic index describing the cohesiveness of a graph, the core number of vertex has been broadly utilized in graph analytics. Specifically, in a graph G, the k-core is the connected subgraph in G, such that each vertex in the subgraph has at least k neighbors. The core number of a vertex v is then defined as the largest k such that there exists a k-core containing v. The parameter of core number is also extensively used in a large number of other applications, to analyze the structure of a network, such as analyzing the topological structure of Internet [8], identifying influential spreader in complex networks [18], analyzing the structure of large-scale software systems [20], [25], predicting the function of biology network [4], and visualizing large networks [2], [29] and so on.

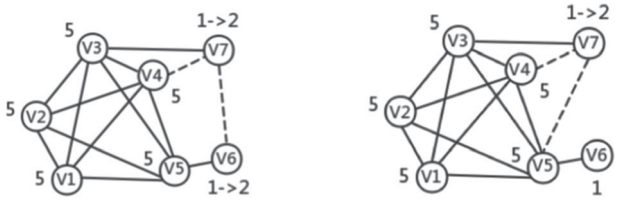
In static graphs, the computation of the core number of each vertex is known as the k-core decomposition problem, which has been extensively studied. The state-of-the-art algorithm is the one proposed in [6]. It can compute the core number of each vertex in $O(m)$ time and m is the number of edges in the graph. However, in many real-world applications, graphs are changing continuously, due to edge/vertex insertions/deletions. In such dynamic graphs, many applications need to maintain the core number for each vertex in real-time. Hence, it is very necessary to study the core maintenance problem, i.e., update the core numbers of vertices in dynamic graphs.

An intuitive way to solve the core maintenance problem is recomputing the core numbers of vertices after every change of the graph. But clearly, this manner is too expensive in large-scale graphs where there might be billions of vertices and trillions of edges. Another manner is just to find the set of vertices whose core numbers will be definitely changed and then update the core numbers of these vertices. However, this manner faces several challenges. First, the exact change value of core number of a vertex is not easy to determine, even if the same number of edges are inserted to a vertex, as shown in Fig. 1. Second, the set of vertices that will change the core number after a graph change is also hard to identify. As shown in Fig. 2, the core numbers of all vertices may change even if only one edge is inserted to the graph.

Due to the great challenge posed in solving the core maintenance problem in dynamic graphs, previous works all focus on the case that only one edge is inserted into/deleted from the graph. In this scenario, it is easy to check that the core number of each vertex can be changed by at most 1. Hence, the first challenge discussed above is avoided, and it only needs to overcome the second difficulty. When multiple edges are inserted/deleted, these edges are processed sequentially, and the core numbers of vertices are updated after each execution of the single-edge update algorithm. However, the sequential processing approach incurs extra overheads when multiple edges are inserted/deleted, since it may unnecessarily repeatedly visit a vertex, as shown in Fig. 3. And on the other hand, it does not make full use of the multi-core machine or distributed systems. Therefore, one natural question is whether we can investigate the parallelism in the edge processing procedure

• *The authors are with the Services Computing Technology and System Lab, Big Data Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, 1037 Luoyu Road, Wuhan 430074, P.R. China. E-mail: {hjin, Ice_lemon, dxyu, qshua, xhshi, shelicy}@hust.edu.cn.*

Manuscript received 8 June 2017; revised 19 Feb. 2018; accepted 18 Apr. 2018. Date of publication 11 May 2018; date of current version 10 Oct. 2018. (Corresponding author: Dongxiao Yu.)
Recommended for acceptance by Umit V. Catalyurek.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2018.2835441



(a) The core number change of v_7: from 1 to 2

(b) The core number change of v_7: from 1 to 3

Fig. 1. In (a) and (b), the number aside the vertices is the core number, two edges are inserted to vertex v_7 respectively, and the core number changes of v_7 in these two cases are 1 and 2 respectively.

and devise parallel algorithm that suits to implement in multi-core machine or distributed systems. In this paper, we answer this question affirmatively by proposing parallel algorithms for core maintenance.

The core maintenance in the scenarios of vertex insertion/deletion can be solved using algorithms for edge insertions/deletions. Take vertex insertion as an example. By setting the initial core number of the inserted vertex as 0 and executing an algorithm for edge insertion to handle the inserted edges generated by the inserted vertices, the core maintenance with respect to vertex insertion can be solved. Therefore, in this paper, we focus on the scenarios that there are only edges inserted into/deleted from the graph. Specifically, the core maintenance problem with respect to edge insertions and deletions are called the *incremental* and *decremental* core maintenance respectively.

Our parallel algorithms are inspired by the single-edge insertion/deletion algorithms, such as those in [19]. To overcome the two difficulties discussed before, we first study the available set of edges whose insertions/deletions only make the core numbers of vertices change by at most one. If the feature of an available set can be determined, then when such an available set of edges are inserted/deleted, each edge can be processed in parallel. We can find the set of vertices whose core number changes is due to a particular edge in the available set. Then the union of these sets of vertices are just those that will change core numbers after inserting the available set of edges, as inserting/deleting the available set can make each vertex change the core number by at most one. Based on this idea, we devise parallel algorithms consisting of two main steps: 1) split the inserted/deleted edges into multiple available sets, and 2) identify the vertices whose core numbers change after inserting/deleting each particular available set and update the core number of the vertices.

Our contributions are summarized as follows.

- We show that if a matching (a set of edges in which any pair do not have common endpoints) is inserted/deleted, the core number of each vertex can change by at most one.
- Based on the structure of matching, we present parallel algorithms for incremental and decremental core maintenance respectively. Our matching based algorithms can reduce the iterations needed significantly, because we can deal with a matching other than one edge in a iteration. When the number of inserted/deleted edges is m_c, the sequential approach will need m_c iterations since it deals with the edges one by one. However in our matching based algorithm, the number of iterations needed is only $\Delta_c + 1$ where Δ_c denotes the maximum number of inserted/deleted edges connecting to a vertex, since a matching can contain an edge connected to each vertex that have edges inserted/deleted. Though the number of inserted/deleted edges can be large, it is still very small in contrast with the number of vertices in a large-scale graph. Hence, Δ_c is small in real-world cases. Our algorithms will provide good parallelism in reality.
- We then conduct extensive experiments on real-world, synthetic graphs and temporal networks, to evaluate the efficiency, stability, scalability and parallelism of the proposed algorithms. The experiment results show that our algorithms exhibit good stability and scalability. Especially, our algorithms achieve better efficiency in handling graph changes of large size. Comparing with sequential algorithms, our algorithms speed up the core number update process on all datasets. In large-scale graphs, such as LiveJournal and Orkut (refer to Table 1 in Section 7), the speedup ratio can be up to 3 orders of magnitude when handling the insertion/deletion of 20000 edges.

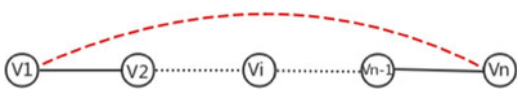


Fig. 2. After the insertion of edge $<v_1, v_n>$, all vertices increase the core number by one.

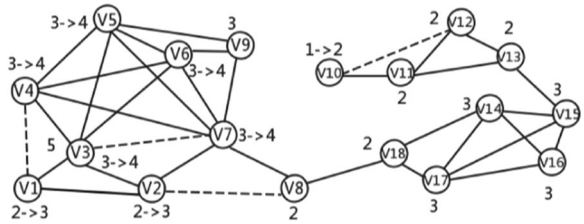


Fig. 3. Assume $<v_1, v_4>, <v_2, v_8>, <v_3, v_7>, <v_{10}, v_{12}>$ will be inserted into the graph. *TRAVERSAL* algorithm in [24] handle the inserted edges one by one. First for edge $<v_1, v_4>$, it will visit vertices v_1, v_2 and update core numbers for v_1, v_2 from 2 to 3. Then inserting $<v_3, v_7>$, it will visit $v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9$, and update core numbers of v_3, v_4, v_5, v_6, v_7 from 3 to 4. The same goes for $<v_2, v_8>$ and $<v_{10}, v_{11}>$. During the process, v_1 and v_2 will be visited for multiple times, but their core numbers are only changed for once. However in our parallel algorithm, all four edges can be processed in parallel using three processors, and duplicate visiting of v_1, v_2 can be avoided.

The rest of this paper is organized as follows. In Section 2, we briefly review related works. In Section 3, the problem definitions are given. Theoretical results supporting the algorithm design are presented in Section 4. The incremental and decremental parallel algorithms are proposed in Sections 5 and 6 respectively. In Section 7, the experiment results are illustrated and analyzed. The whole paper is concluded in Section 8.

2 Related Work

Finding cohesive structures in a graph is of much importance in graph analytics. There exist a lot of definitions for cohesiveness of a graph, such as cliques, k-truss, k-core, F-groups, n-clans and so on [14]. Among them k-core is recognized as one of the most efficient and helpful ones, since

most of the measurements have a computational complexity of quadratic or even NP-hard, while k-core can be computed in linear time complexity. In this section, we make a brief review of related works on core decomposition and core maintenance below.

Core Decomposition. In static graphs, the core decomposition problem, which is to compute the core numbers of vertices, have been widely studied. In [6], an $O(m)$ time algorithm was presented, where m is the number of edges in the graph. This result is the state-of-the-art one. In [10], Cheng et al. proposed an external-memory algorithm called *EMcore* when the graph is too large to be held in memory. In [22], Montresoret al. considered the problem in a distributed situation, and they proposed an algorithm that can be applied to both one-to-one and one-to-many computational models. In [17]. Khaouid et al. conducted an investigation of core decomposition in a single PC and compared the above three solutions using GraphChi and WebGraph models. In addition, parallel core decomposition was studied in [11], and an algorithm called *ParK*, which can reduce the working set size and minimize the random accesses, was proposed.

Core Maintenance. Core maintenance is to update the core numbers instead of recomputing them when the graph changes. In [24], Sariyuce discovered that when an edge is inserted/deleted the core number of any vertex in the graph can change by at most 1. Based on this observation, a linear algorithm named *TRAVERSAL* was proposed to identify vertices that change core numbers due to the inserted/deleted edges. Concurrently, similar findings were given in [19], however the solution in [19] is quadratic complexity. In [26], how to improve the I/O efficiency was studied, when computing and maintaining the core numbers of vertices. Distributed solutions for core maintenance of single-edge change were studied in [1] and [3]. But as described before, there exist lots of unnecessarily repeated computations which lead to deficiency. In [30], the authors proposed an algorithm to maintain core numbers based on the orders between vertices in core decomposition of static graph. All above works focus on solving the problem of core maintenance when only one edge is inserted into/deleted from the graph. In [28], a parallel algorithm using a similar framework as that in this paper was proposed. Specifically, a superior edge structure was proposed for parallel processing. Compared with the matching structure used in this work, the superior edge structure has to be computed before each iteration, while the matching structures can all be found before the processing of edges, using a simple coloring algorithm on inserted/deleted edges. Hence, the utilization of matching structure can greatly reduce the preprocessing time.

3 Problem Definitions and Preliminaries

We consider an undirected, unweighted simple graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. Let $n = |V|$ and $m = |E|$. For a vertex $u \in V$, the set of its neighbors in G is denoted as $N(u)$, i.e., $N(u) = \{v \in V | (v, u) \in E\}$. The number of u's neighbors in G is called the degree of u, denoted as $d_G(u)$. So $d_G(u) = |N(u)|$. The maximum and minimum degree of nodes in G is denoted as $\Delta(G)$ and $\delta(G)$ respectively.

We next give formal definitions for the *core number* of a vertex and other related concepts.

Definition 1 (k-Core). *Given a graph $G = (V, E)$ and an integer k, the k-core is a connected subgraph H of G, in which each vertex has at least k neighbors, i.e., $\delta(H) \geq k$.*

Definition 2 (Core Number). *Given a graph $G = (V, E)$, the core number of a vertex $u \in G$, denoted by $core_G(u)$, is the the largest k, such that there exists a k-core containing u. For simplicity, we use $core(u)$ to denote $core_G(u)$ when the context is clear.*

Definition 3 (Max-k-Core). *The max-k-core associated with a vertex u, denoted by H_u, is the k-core with $k = core(u)$.*

In this work, we aim at maintaining the core numbers of vertices in dynamic graphs. Specifically, we define two categories of graph changes: *incremental*, where a set of edges E' are inserted to the original graph, and *decremental*, where a set of edges are deleted. Based on the above classification, we distinguish the core maintenance problem into two scenarios, as defined below.

Definition 4 (Incremental Core Maintenance). *Given a graph $G = (V, E)$, the incremental core maintenance problem is to update the core numbers of vertices after an incremental change to G.*

Definition 5 (Decremental Core Maintenance). *Given a graph $G = (V, E)$, the decremental core maintenance problem is to update the core numbers of vertices after a decremental change to G.*

The *core number* of an edge is defined as the smaller value of the core numbers of its endpoints. A set of edges E' is called a *matching*, if for each pair of edges in E', they do not have common endpoints. If all edges in a matching have the same core number k, the matching is called a *k-core matching*.

We next give some notations that help identify the set of vertices which will change core numbers after graph change. Given a graph $G = (V, E)$, let G' be the graph obtained after inserting/deleting an edge set E' into/from G.

Definition 6 (Superior Degree). *For a vertex $u \in V$, v is a* superior neighbor *of u if v is a neighbor of u in G' and $core_G(v) \geq core_G(u)$. The number of u's superior neighbors is called the* superior degree *of u, denoted as $SD_{G'}(u)$.*

The SD value of a vertex u represents the number of neighbors that have a core number no less than $core_G(u)$. It is easy to see that only superior neighbors of u may affect its core number change.

Definition 7 (Constraint Superior Degree). *The* constraint superior degree *$CSD_{G'}(u)$ of a vertex u is the number of u's neighbors w in G' that satisfies $core_G(w) > core_G(u)$ or $core_G(w) = core_G(u) \wedge SD_{G'}(w) > core_G(u)$.*

The constraint superior degree of a vertex u counts for two categories of neighbors that will affect the increase of u's core number: the ones that have larger core numbers and those that have the same core number but have enough neighbors which may make them increase the core number. When the context is clear, we use $SD(u)$ and $CSD(u)$ to present the SD and CSD values of vertex u in the current new graph. Note that the definition of SD and CSD are similar to the MCD and PCD defined in [24]. The SD value can be used to

identify if a vertex will change its core number after an edge insertion/deletion. The following lemmas were given in [24].

Lemma 1 ([24]). *For a vertex u with core number k, if $SD_{G'}(u) < k$, then u will decrease the core number.*

For the deletion case, a similar result was also obtained in [24]. For a vertex u, if $CSD_{G'}(u) \leq core_G(u)$, then u cannot increase its core number, since it does not have enough support neighbors. This condition is summarized formally as below and will be used to determine whether a vertex is impossible to increase the core number.

Lemma 2. *For a vertex u, if $CSD_{G'}(u) \leq core_G(u)$, u will not increase its core number.*

4 Theoretical Basis

In this section, we present some theoretical results that constitute the basis of our parallel algorithms. In particular, we first show that the core number of every vertex can change by at most one, if the inserted/deleted edges form a matching. And then we depict the feature of vertices whose core numbers will change after inserting/deleting a matching.

4.1 Core Number Change of Vertices After Insertion/Deletion of a Matching

The main results are given in Lemmas 4 and 5 below. At first, we prove the following Lemma 3, which is useful in proving Lemmas 4 and 5.

Lemma 3. *For a k-core $H = (V_H, E_H)$, $\delta(H) = k$, if after deleting a set of edges, H becomes H', where H' is still connected, and the degree of each vertex in H is decreased by at most 1, then H' will be a k-1-core.*

Proof. After deleting the edges, it can be concluded that $\delta(H') \geq k - 1$. Then the result is obtained by Definition 1. □

Lemma 4. *Given a graph $G = (V, E)$, if a matching $E_M = \{e_1, e_2, ..., e_d\}$ is inserted into G, then the core number of every vertex $w \in V$ can increase by at most 1.*

Proof. First we assume that after the insertion, a vertex w with core number k increases the core number to $k + x$, where $x > 1$. We denote the max-$(k + x)$-core of w after the insertion is H_w^+ and the max-k-core of w before insertion is H_w, then $\delta(H_w^+) = k + x$ and $\delta(H_w) = k$. It must be true that at least one of the inserted edges $e_i \in H_+^w$ for $1 \leq i \leq d$, as otherwise $\delta(H_w) = k + x$ and $core_G(w) = k + x$, which is a contradiction. Let $Z = H_w^+ \setminus E_M$, then $Z \subset G$. If Z is connected, Z will be a $k + x - 1$-core by Lemma 3, since the degree of every vertex in H_w^+ decreases by at most 1 due to the removal of a matching. This leads to a contradiction, since $k + x - 1 > k$ and H_w is max-k-core in G. If Z is disconnected, each connected component will be a $k + x - 1$-core by Lemma 3, since the degree of every vertex in each connected component decreases by at most 1 due to the removal of edges in E_M. This also leads to a contradiction. The result is then proved. □

Using a similar argument as that for proving Lemma 4, we can get the result for deletion of a matching, as shown in the following Lemma 5.

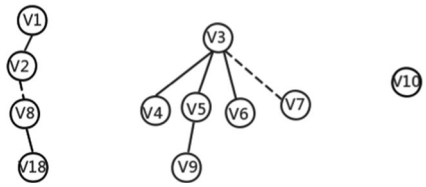


Fig. 4. For the graph in Fig. 3, after inserting the matching E_m, the *exPath-Tree* of E_m consists of K-Path-Tree of vertex v_1, v_3, v_{10}. Notice that v_1, v_2 are in the same tree.

Lemma 5. *Given a graph $G = (V, E)$, if a matching $E_M = \{e_1, e_2, ..., e_d\}$ is deleted from G, then for every $w \in G$, $core(w)$ can decrease by at most 1.*

4.2 Identifying Vertices with Core Number Changes After Insertion/Deletion of a Matching

We next identify the set of vertices whose core numbers change after a matching is inserted into/deleted from the graph. Some notations will be first defined. Consider the scenario defined as follows: given a graph $G = (V, E)$ and an edge set $E_s = \{e_1, e_2, ..., e_s\}$, w.l.o.g., assume that for each $e_i = <u_i, v_i>$, $core_G(v_i) \geq core_G(u_i) = k_i$, where $s > 0$, $1 \leq i \leq s$, and $k_i \geq 1$. After E_s is inserted into or deleted from G, G becomes a new graph $G' = (V, E')$.

Next, we define the K-Path-Tree and exPath-Tree in the following Definitions 8 and 9. As shown later, after inserting/deleting a matching, only vertices on the exPath-Tree will change the core number.

Definition 8 (*K*-Path-Tree). *For the new graph $G' = (V, E')$, $\forall u \in V$, assume $core_G(u) = k$, the* K-Path-Tree *of u is a tree rooted at u and each vertex w in the tree satisfies $core_G(w) = core_G(u)$. For simplicity we use $T_k^{G'}(u)$ to represent the K-Path-Tree of u in G'.*

Basically, the definition of *K-Path-Tree* is similar as the subcore defined in [24], but notice that *K-Path-Tree* is defined on the new graph G', while the subcore in [24] is defined on the original graph. Besides we use the core number of vertices in the original graph to identify *K-Path-Tree*.

Definition 9 (exPath-Tree). *For the new graph $G' = (V, E')$ and the edge set E_s, the union of K-Path-Tree for every u_i is called the* exPath-Tree *of E_s, denoted as $exPT_{G'}(E_s)$.*

Fig. 4 shows an example of the *K-Path-Tree* and *exPath-Tree*.

We first consider the case of inserting/deleting edges in a k-core matching.

Lemma 6. *Given a graph $G = (V, E)$, if a k-core matching $E_k = \{e_1, e_2, ..., e_p\}$ with $|E_k| = p$, is inserted into or deleted from G, where $k \geq 1$, we have*

- *(i) only vertices in $exPT_{G'}(E_k)$ may change the core number, where G' is the graph obtained from G after inserting/deleting E_k;*
- *(ii) the change is at most 1.*

Proof. (ii) can be easily gained by Lemma 4. We next prove (i).

In the insertion case, we first prove that if a vertex $w \in V$ such that $core_G(w) = x \neq k$, w will not change its core number. Assume $core_G(w)$ increases to $x + 1$, and we

denote the max-k-core of w after and before insertion as H_u^+ and H_u, respectively. We know that H_w is a x-core and H_w^+ is a $x+1$-core. Then at least one of the new edges must belong to H_w^+, as otherwise H_w would be a $x+1$-core before insertion. Assume $e_i = <u_i, v_i> \in H_w^+$. Then we have $k+1 \geq x+1$ since $e_i \in H_w^+$ and $core_{G'}(u_i) \leq k+1$. This implies that $x < k$. Hence, $H_w^+ \setminus E_k$ is a $x+1$-core before insertion, which is a contradiction. So if a vertex has a core number not equal to k, it will not change its core number.

Then we prove that if a vertex $w \in V$ such that $core_G(w) = k$ but $w \notin exPT_{G'}(E_k)$ will not change its core number. Assume w increases its core number to $k+1$. We can have that w must either have a new neighbor or at least one of its neighbors increases core number. Obviously w do not have any new neighbor and we have proved that vertices whose core number is not equal to k will not increase the core number. So w must have a neighbor whose core number is k and increases core number. Applying this recursively, we will finally reach a vertex u whose core number change is caused by a new neighbor, and vertices on the recursive path all have a core number k, say, w is in the $T_k^{G'}(u)$, which is a contradiction.

We have proved that for a vertex w, if $core_G(w) \neq k$ or $core_G(w) = k$ but $w \notin exPT_{G'}(E_k)$ will not change its core number, so only vertices in the $exPT_{G'}(E_k)$ may have their core numbers increased. The proof for the insertion case is completed.

The deletion case can be proved similarly as above. □

Based on Lemma 6, we next consider the case of inserting/deleting a matching.

Lemma 7. *Given a graph $G = (V, E)$, if a matching $E_M = \{e_1, e_2, ..., e_m\}$ is inserted into or deleted from G, then we can have that only vertices w on $exPT_{G'}(E_M)$ can change the core number and the change is at most 1.*

Proof. It is easy to see that inserting/deleting a matching E_M can have the same result as inserting/deleting the k-core matchings in E_M one by one. Assume $E_M = E_{k_1} \cup E_{k_2} \cup, ..., \cup E_{k_p}$, E_{k_i} is a k_i-core matching and $k_i < k_j$ where $1 \leq i < j \leq p$. We insert/delete the k_i-core matchings one by one, and denote the graph after inserting/deleting E_{k_i} as G_{k_i}.

First for the insertion case, when inserting E_{k_i}, by Lemma 6, we can know that, only vertices on $exPT_{G_{k_i}}(E_{k_i})$ can increase the core number by at most 1. Here notice that the core numbers of some vertices on $exPT_{G_{k_i}}(E_{k_i})$ may be $k_i - 1$ in $G_{k_{i-2}}$ and increase by 1 to k_i after inserting $E_{k_{i-1}}$. We denote these vertices as $B_{k_{i-1}}$. To prove our result, we need to show that vertices in $B_{k_{i-1}}$ will not increase the core number any more. This can be obtained by Lemma 4, since every vertex can change the core number by at most 1 after the insertion of a matching.

The deletion case can be proved using a similar argument as above. The proof is completed. □

Summary. In this section, we showed that if a matching E_M is inserted into or deleted from the graph G, G becomes G', then only vertices in the $exPT_{G'}(E_M)$ can change their core number by at most 1. Besides, only vertices satisfying $CSD_{G'}(u) > core_G(u)$ for insertion case and satisfying $SD_{G'}(u) < core_G(u)$ for deletion case will change the core number. Our algorithms in the following sections rely on these results.

5 Incremental Core Maintenance

In this section, we present the parallel algorithm for incremental core maintenance after inserting an arbitrary edge set E_I to graph $G = (V, E)$. Let V_I denote the set of vertices connecting to edges in E_I. We define the *maximum insertion degree* Δ_I as the maximum number of edges inserted to each vertex in V.

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

Input
The graph, $G = (V, E)$;
The inserted edge set, E_I;
The core number $core(v)$ of each vertex in G;
Output
The new core number of each vertex;
Initially $\mathcal{C} \leftarrow$ empty core set;
1 Properly color the inserted edges in E_I using at most $\Delta_I + 1$ colors $\{1, 2, \ldots, \Delta_I + 1\}$ by executing the coloring algorithm in [21];
2 $max_c \leftarrow$the max color of edges in E_I;
3 $c \leftarrow 1$, $G_0 \leftarrow G$;
4 **while** $c \leq max_c$ **do**
5 reset $\mathcal{C}$ to be an empty core set;
6 $E_c \leftarrow$ edges in E_I with color c;
7 $G_c = G_{c-1} \cup E_c$;
8 **for** *each edge* $e \in E_c$ **do**
if *core number of* e *is not in* $\mathcal{C}$ **then**
add core number of e to $\mathcal{C}$;
9 **for** *each core number* k *in* $\mathcal{C}$ *in parallel* **do**
10 $E_k \leftarrow$ edges in E_c with core number k;
$V_k \leftarrow K$-Core-MatchingInsert($G_c, E_k, core()$);
11 **for** *each vertex* $v \in \cup_{k \in \mathcal{C}} V_k$ **do**
$core(v) \leftarrow core(v) + 1$;
12 $c \leftarrow c + 1$;
13 **return** core();

Algorithm. The detailed algorithm is given in Algorithm 1. The algorithm consists of two main steps. At first, a pre-processing procedure is executed, to split the edges into several matchings. Then the algorithm is executed in iterations in each of which a matching is inserted into the graph, and the set of vertices that change core numbers are found.

More specifically, the pre-processing is done by properly coloring the inserted edges, such that any pair of edges with common endpoint receive different colors. It is easy to see that the edges with the same color constitute a matching.

In the second step, as shown in Lemma 5, when inserting a matching into the graph, the core number of every vertex can increase by at most one. This means that we only need to find the set of vertices that increase core numbers because of the insertion of each particular edge, and the union of these vertices is just the set of vertices which will increase the core number by one. Hence, in each iteration, the edges are processed in parallel. But it deserves to pointing out that we do not make each edge processed by a processor. Instead, the edges with the same core number in the inserted matching are processed on a processor. This is to decrease duplicated processing of vertices. By Lemma 6, the

edges in a k-core, i.e., the edges with core number k in the inserted matching, affect the core number change of the same set of vertices, those on $exPT_{G_c}(E_k)$, where E_k is the k-core matching. And by Lemma 5, each vertex can increase the core number by at most one. This means that when a vertex is determined to increase its core number, after inserting an edge in E_k, it is unnecessary to visit it any more when inserting other edges in E_k. Hence, by processing the edges with the same core number on a process can greatly decrease unnecessarily repeated visiting to vertices.

Algorithm 2. K-Core-MatchingInsert(G_c, E_k, core())

Input
The current new graph, $G_c = (V, E_c)$;
The inserted edges with core number k, E_k;
The current core number $core(v)$ of each vertex v;
Output
Vertices that will increase core numbers;
Initially, $S \leftarrow$ empty stack, $V_c \leftarrow$ empty vertex set;
$\forall v \in V$,
$visited[v] \leftarrow false, removed[v] \leftarrow false, cd[v] \leftarrow 0$;
1 compute SD value for each vertex v in $exPT_{G_c}(E_k)$;
2 **for** each $e_i = <u_i, v_i> \in E_k$ **do**
3 **if** $core(u_i) \geq core(v_i)$ **then**$r \leftarrow v_i$
else $r \leftarrow u_i$
4 $k \leftarrow core(r)$;
5 **if** $visited[r]$ = *false and* $removed[r]$ = *false* **then**
6 **if** $CSD[r] = 0$ **then** compute $CSD[r]$
if $cd[r] >= 0$ **then** $cd[r] \leftarrow CSD[r]$ **else**
$cd[r] \leftarrow cd[r] + CSD[r]$
$S.push(r)$;
$visited[r] \leftarrow true$;
7 **while** *S is not empty* **do**
$v \leftarrow S.pop()$;
8 **if** $cd[v] > k$ **then**
9 **for** *each* $< v, w > \in E_c$ **do**
10 **if** $core(w) = k$ *and* $SD(w) > k$ *and*
$visited[w]$ = *false* **then**
$S.push(w)$;
$visited[w] \leftarrow true$;
if $CSD[w] = 0$ **then**
compute $CSD[w]$;
$cd[w] \leftarrow cd[w] + CSD[w]$;
11 **else**
if $removed[v]$ = *false* **then**
InsertRemove($G_c, core(), cd[]$,
$removed[], k, v$)
12 **for** *each vertex* v *in* G **do**
if $removed[v]$ = *false and* $visited[v]$ = *true* **then**
$V_c \leftarrow V_c \cup \{v\}$
13 **return** V_c;

Algorithm 2 is used to search the vertices with core number changes when inserting a k-core matching E_k. Algorithm 2 first computes SD values for vertices on $exPT_{G_c}(E_k)$, then handles the insertion of these edges one by one. In particular, for each edge $e_i = <u_i, v_i> \in E_k$, it conducts a DFS search from the root which is the one in $\{u_i, v_i\}$ with smaller core number (if $core(u_i) = core(v_i)$, the tie is broken arbitrarily). Each vertex is maintained with a value cd which counts the number of neighbors that may help it increase the core number. The initial value of cd of a vertex is set as its constraint superior degree. Vertices in $exPT_{G_c}(E_k)$ that are connected to the root are pushed into the stack, if they satisfy that the SD value is larger than k. Only these vertices are possible to increase the core number by the definition of SD. Then we use the condition that $cd \leq k$ to determine that a particular vertex is impossible to increase the core number. Specifically, at every time, a vertex v is fetched from the stack. If $cd[v] > k$ which means v may be in a k+1-core, its neighbors are then visited. Otherwise, v cannot be in a k+1-core. An algorithm given in [24] is conducted to spread the influence to other vertices and update the cd values of other vertices. The details are shown in Algorithm 3.

Algorithm 3. InsertRemove($G_c, core(), cd[], removed[], k, r$)

Input
The current new graph, $G_c = (V, E_c)$;
The $core(v), cd[v], removed[v]$ of each vertex;
The core number k and root r;
1 $S \leftarrow$ *empty stack*;
2 $S.push(r)$, $removed[r] \leftarrow true$;
3 **while** *S is not empty* **do**
$v \leftarrow S.pop()$;
for *each* $< v, w > \in E_c$ **do**
if $core(w) = k$ **then**
4 $cd[w] \leftarrow cd[w] - 1$;
if $cd[w] = k$ *and* $removed[w]$ = *false* **then**
5 $S.push(w)$;
6 $removed[w] \leftarrow true$;

Performance Analysis. In this part, we analyze the correctness and efficiency of our incremental algorithm. To depict the complexity of the algorithm, we first define some notations.

For graph $G = (V, E)$, the inserted edge set E_I and a subset S of E_I, let $G_S = (V, E \cup S)$ and $K(G_S)$ be the set of core numbers of vertices in G_S.

For G_S, let $L_S = \max_{S' \subset E_I \setminus S} \max_{u \in V} \{CSD_{G_{S \cup S'}}(u) - core_{G_S}(u), 0\}$. As shown later, L_S is the maximum times a vertex u can be visited by $InsertRemove$ procedure during the iteration when inserting edges to G_S.

For a $k \in K(G_S)$, let $V_S(k)$ be the set of vertices with core number k, and $N(V_S(k))$ be the neighbors of vertices in $V_S(k)$. Let $n_S = \max\{|V_S(k)| : k \in K(G_S)\}$.

Denoted by $E[V_S(k)]$ the set of edges in G_S that are connected to vertices in $V_S(k) \cup N(V_S(k))$. Then we define a parameter m_S as follows, which represents the maximum number of edges travelled when computing SD after inserting edges to G_S.

$$m_S = \max_{k \in K(G_S)} \{|E[V_S(k)]|\}.$$

With the above notations, we prove the correctness and bound the running time of our algorithm in the following Theorem 8.

Theorem 8. *Algorithm 1 can update the core numbers of vertices in* $O(|E_I| * |V_I| + \Delta_I * \max_{S \subseteq E_I}\{m_S + L_S * n_S\})$ *time, after inserting an edge set* E_I *to a graph* G*, where* V_I *is the set of vertices in* G *connecting to edges in* E_I *and* Δ_I *is the maximum insertion degree.*

Proof. We first prove the correctness. As discussed before, a proper coloring can split the inserted edges into multiple

matchings. When inserting a matching into the graph, it is only necessary to find the vertices that change their core numbers and increase their core numbers by one. Hence, the edges in a matching can be processed in parallel. This constitutes the base of our parallel algorithm. Then when processing a particular matching E_k in a process, by Lemmas 6 and 7, only vertices in $exPT_{G_c}(E_k)$ are possible to increase the core number. Hence, Algorithm 2 visit all vertices that may change the core number. Furthermore, based on the definitions of CSD and cd and Lemma 2, if $cd(v) < k$ for a vertex v, it is impossible for v to increase the core number. Therefore, the algorithm removes vertices that will not increase the core number definitely. And the negative DFS process in Algorithm 3 ensures to spread the influence of a removed vertex, i.e., update the cd values (which count the number of neighbors that may support a particular vertex's core number change) of vertices in $exPT_{G_c}(E_k)$. By Lemma 1, each vertex being determined to increase the core number is unnecessary to visit any more, as shown in Algorithm 2, since these vertices will not increase their core numbers any more. Finally, after executing the algorithm, vertices that are visited but not removed will increase their core numbers, as these vertices all have at least $k+1$ neighbors and they constitute $k+1$-cores. Combining the above together, the correctness of the algorithm is guaranteed.

As for the time complexity, there are two stages in the algorithm: first, the inserted edges are split into matchings by coloring, and second, in each iteration, edges in a matching with the same core number are inserted to the graph and the core numbers of vertices are updated. As shown in [21], the coloring of inserted edges takes $O(|E_I| * |V_I|)$ time. We next analyze the time used in the second stage.

We first bound the number of iterations. In each iteration, the edges with one particular color are processed. As shown in [21], the number of colors used is at most $\Delta_I + 1$. Hence, the number of iterations is also upper bounded by $\Delta_I + 1$.

Now consider an iteration i in the second stage of the algorithm execution. Denote by G_i the graph obtained after iteration $i-1$ as G_i, and by E_M the matching gained in iteration i. The computation of CSD values for vertices in $exPT_{G_c}(E_M)$ takes $O(m_{E_M})$ time. In the algorithm, each vertex is visited for once to determine whether to update its core number. But it needs to notice that each vertex may be visited for multiple times in the negative DFS procedures that disseminate the influence of a removed vertex. However, if a vertex v is visited in the negative DFS procedure, $cd(v)$ is decreased by 1. Hence, each vertex can be visited by at most L_S times, since a vertex will be removed if its cd value is decreased to its core number. Then we can get that the total time for an iteration is upper bounded by $O(m_{E_M} + L_{E_M} * n_{E_M})$.

By all above, the running time of the whole algorithm can be bounded as stated. □

6 Decremental Core Maintenance

The algorithm for decremental core maintenance, as given in Algorithm 4, is very similar to the incremental one. The only difference is that we use Lemma 1 instead of Lemma 2 to determine whether a vertex will decrease the core number after deleting edges. The maximum number of edges deleted from each vertex is defined as the *maximum deletion degree*, denoted as Δ_D.

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

Input
The graph, $G = (V, E)$;
The deleted edge set, E_D;
The core number $core(v)$ of each vertex in G;
Output
The new core number of each vertex;
Initially $\mathcal{C} \leftarrow$ empty core set;
1 Properly color the deleted edges in E_D using $\Delta_D + 1$ colors $\{1, 2 \ldots, \Delta_D + 1\}$ by executing the coloring algorithm in [21];
2 $max_c \leftarrow$ the max color of edges in E_D;
3 $c \leftarrow 1, G_0 \leftarrow G$;
4 **while** $c < max_c$ **do**
5 $E_M \leftarrow$ edges in E_D with color c;
6 $G_c = G_{c-1} \setminus E_M$;
7 **for** *each edge* $e \in E_M$ **do**
if *core number of* e *is not in* $\mathcal{C}$ **then**
add core number of e to $\mathcal{C}$;
8 **for** *each core number* k *in* $\mathcal{C}$ *in parallel* **do**
9 $E_k \leftarrow$ edges in E_M with core number k;
$V_c \leftarrow K$-Core-MatchingDelete($G_c, E_k, core()$);
10 **for** *each vertex* v *in* $\cup_{k \in \mathcal{C}} V_c$ **do**
$core(v) \leftarrow core(v) - 1$;
11 $c \leftarrow c + 1$;
12 **return** core();

Performance Analysis. The correctness and efficiency of the proposed decremental algorithm can be analyzed similarly as the incremental one. At first, we define some notations.

For graph $G = (V, E)$, the deleted edge set E_D and a subset R of E_D, let $G_R = (V, E \setminus R)$ and $K(G_R)$ be the set of core numbers of vertices in G_R.

For G_R, let $F_R = \max_{R' \subset E_D \setminus R} \max_{u \in V} \{SD_{G_{R \cup R'}}(u) - core_{G_R}(u) + 1, 0\}$.

For $k \in K(G_R)$, let $V_R(k)$ be the set of vertices with core number k and $n_R = \max\{|V_R(k)| : k \in K(G_R)\}$.

Denote by $E(V_R(k))$ the set of edges connected to vertices in $V_R(k)$. We then define m_R as follows,

$$m_R = \max_{k \in K(G_R)} \{|E(V_R(k))|\}.$$

F_R, n_R and m_R will depict the time used in each iteration in the algorithm execution.

Using a similar argument as that for analyzing the incremental algorithm, we can get the following result, which states the correctness and efficiency of the decremental algorithm.

Theorem 9. *Algorithm 4 can update the core numbers of vertices in* $O(|E_D| * |V_D| + |\Delta_D * \max_{R \in E_D}\{m_R + F_R * n_R\})$ *time, after deleting an edge set* E_D *from a graph* G*, where* V_D *is the set of vertices in* G *connecting to edges in* E_D *and* Δ_D *is the maximum deletion degree.*

Proof. We prove the correctness first. Similarly as the insertion case, the coloring algorithm split the deleted edges into multiple matchings, and edges in a matching can be

TABLE 1
Real-World Graph Datasets

Datasets	n = $\|V\|$	m = $\|E\|$	max degree	max core
AP(ca-Astroph)	18.7K	198.1K	504	56
S1(soc-Slashdot)	82.1K	500.5K	2548	54
DB(DBLP)	0.31M	1.01M	343	113
GW(Gowalla)	196.6K	1.9M	14730	51
YT(YouTube)	1.13M	5.97M	28754	51
BS(web-BerkStan)	0.68M	13.3M	84230	201
LJ(LiveJournal)	4.0M	34.7M	20334	360
OK(Orkut)	3.0M	117.2M	56382	253

processed in parallel. Then when dealing with a matching E_M, we distribute the k-core matchings in it into several child processes. In a child process, by Lemmas 6 and 7, only vertices in $exPT(E_k)$ may decrease the core number, which is conducted in Algorithm 5. Furthermore, by the definition of SD and cd, we can get that for a vertex v with $core(v) = k$, if $cd(v) < k$, $core(v)$ will decrease. So the algorithm remove the vertices that decrease the core number, which is done in Algorithm 6 through a DFS. Besides, Algorithm 6 updates the cd values of vertices in $exPT(E_k)$. By Lemma 6, vertices that will decrease the core number need not to be visited any more, as their core numbers can only decrease by at most 1. Finally, after executing the algorithm, vertices that have been visited and removed will decrease their core numbers, as these vertices do not have enough neighbors to keep the core number k. Combining the above together, the correctness of the algorithm is guaranteed.

As for the time complexity, there are two stages in the algorithm: first, the deleted edges are split into matching by edge coloring, and second, in each iteration, edges in a matching with the same core number are deleted from the graph and the core numbers of vertices are updated. As shown in [21], the coloring of E_D takes $O(|E_D| * |V_D|)$ time. We next analyze the time used in the second stage.

We first bound the number of iterations. In each iteration, the edges with one particular color are processed. As shown in [21], the number of colors used is at most $\Delta_I + 1$. Hence, the number of iterations is also upper bounded by $\Delta_D + 1$.

Now consider an iteration i in the second stage of the algorithm execution. Denote by G_i the graph obtained after iteration $i-1$ as G_i, and by E_M the matching gained in iteration i. The computation of SD values for vertices in $exPT(E_M)$ takes $O(m_{E_M})$ time. In the algorithm, each vertex may be visited for multiple times in the DFS procedures that disseminate the influence of a removed vertex. However, if a vertex v is visited in the DFS procedure, $cd(v)$ is decreased by 1. Hence, each vertex can be visited by at most F_R times, since a vertex will be removed if its cd value is decreased to be smaller than its core number. Then we can get that the total time for an iteration is upper bounded by $O(m_{E_M} + F_R * m_R)$.

By all above, the running time of the whole algorithm can be bounded as stated. □

7 Experiment Studies

In this section, we evaluate the performances of our algorithms by experiments. The experiments use three synthetic data sets, eight real-world graphs and three temporal networks. The details of the graphs are shown in Tables 1 and 2. The maximal degree and the core number in Table 2 are those of the temporal networks at the last time point.

TABLE 2
Temporal Networks

Datasets	n = $\|V\|$	Temporal Edges	Static Edges	max degree	max core
SU	197K	1.44M	924.9K	14294	61
WK	1.14M	7.8M	3.3M	141951	124
ST	2.6M	63.5M	36.2M	44065	198

Algorithm 5. K-Core-MatchingDelete($G_c, E_k, core()$)

Input
The current new graph, $G_c = (V, E_c)$;
The edges with core number k, E_k;
The core number $core(v)$ of each vertex in V;
Output
Vertices that will decrease core numbers;
Initially, $S \leftarrow$ empty stack, $V_c \leftarrow$ empty vertex set; $\forall v \in V$, $visited[v] \leftarrow false, removed[v] \leftarrow false, cd[v] \leftarrow 0$;

```
1  for each e_i = <u, v> ∈ E_k do
2    if core(u) ≥ core(v) then r ← v;
     else r ← u;
3    k ← core(r);
4    if core(v) ≠ core(u) then
       if visited[r] = false then
         visited[r] ← true;
         cd[r] ← SD(r);
5      if removed[r] = false then
         if cd[r] < k then
6          DeleteRemove(G_c, core(), cd[],
           removed[], k, r)
7  else
8    if visited[u] = false then
       visited[u] ← true;
       cd[u] ← SD[u];
9    if removed[u] = false then
10     if cd[u] < k then
         DeleteRemove(G_c, core(), cd[],
         removed[], k, u)
11   if visited[v] = false then
       visited[v] ← true;
       cd[v] ← SD(v);
12   if removed[v] = false then
       if cd[v] < k then
13       DeleteRemove(G_c, core(), cd[],
         removed[], k, v)
14 for each vertex v in G do
     if removed[v] = true and visited[v] = true then
       V_c ← V_c ∪ {v}
15 return V_c;
```

We first evaluate the efficiency of our algorithms on real-world graphs and temporal networks. Then, we evaluate the scalability of our algorithms using synthetic graphs. At last, we compare our algorithms with the state-of-the-art sequential core maintenance algorithms, TRAVERSAL algorithms given in [24], and the parallel Superior Edge based algorithm proposed in [28], to evaluate the speedup ratio of

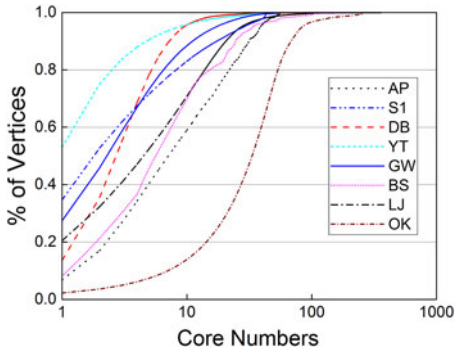


Fig. 5. Core number distribution of real-world graphs.

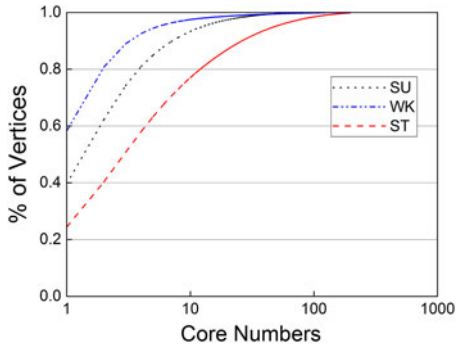


Fig. 6. Core number distribution for temporal graphs.

our algorithms. In the experiments, to simulate the edge insertion/deletions, we first delete some edges from a graph to test the deletion algorithm, and then add the edges back to test the insertion algorithm.

Algorithm 6. DeleteRemove($G_c, core(), cd[], removed[], k, r$)

Input
The current new graph, $G_c = (V, E_c)$;
The $core(v), cd[v], removed[v]$ of each vertex;
The core number k and root r;
$S \leftarrow empty\ stack$;
$S.push(r), removed[r] \leftarrow true$;
while S is not empty **do**
  $v \leftarrow S.pop()$;
  for *each* $< v, w > \in E_c$ **do**
    if $core(w) = k$ **then**
      if $visited(w) =$ *false* **then**
        $visited[w] \leftarrow true$;
        $cd[w] \leftarrow cd[w] + SD(w)$;
      $cd[w] \leftarrow cd[w] - 1$;
      if $cd[w] < k$ *and* $removed[w] =$ *false* **then**
        $S.push(w)$;
        $removed[w] \leftarrow true$;

All experiments are conducted on a Linux machine having 8 Intel Xeon E5-2670@2.60 GHz CPUs with support for 16 concurrent threads and 64 GB main memory. The algorithm is implemented in C++ and compiled with g++ compiler using the -O3 optimization option.We use adjacency list data structure to store the graph. Basically, the neighbors of a vertex are stored in a vector to support efficient insertions and deletions of vertices and edges. The parallelism is implemented through a thread pool which has a fixed number of threads. The thread pool maintains a thread queue and a task queue, and one task is delivered to a thread at one time. When the number of tasks is larger than the number of threads, the threads are reused.

Data Sets. We use eight real-world graphs and random graphs generated by three different models. The eight real-world graphs can be downloaded from SNAP [16], including social network graphs (soc-Slashdot, Youtube, Gowalla, LiveJournal, Orkut), collaboration network graphs (DBLP, ca-Astroph) and Web graphs (web-BerkStan). The synthetic graphs are generated by the SNAP system using three models: the Block Two-Level Erdös-Rényi (BTER) graph model [13], which generates a random graph; the Barabasi-Albert (BA) preferential attachment model [5], where each vertex has k connected edges; and the R-MAT (RM) graph model [9], which can generate large-scale realistic graphs similar to social networks. For all generated graphs, the average degree is fixed to 8, such that when the number of vertices in the generated graphs is the same, the number of edges is the same as well. WK, SU, ST are three temporal networks where each edge has a timestamp. Wiki-talk (WK) temporal network represents Wikipedia users editing each other's talk page. Super User (SU) temporal network is a temporal network of interactions on the stack exchange web site Super User. Stack Overflow (ST) temporal network is a temporal network of interactions on the stack exchange web site Stack Overflow.

Figs. 5, 6 and 7 show the core number distributions of the eight real-world graphs, the temporal networks (at the last time point) and the generated graphs with 2^{21} vertices respectively. From Figs. 5 and 6, it can be seen that in both real-world graphs and temporal networks, most of the vertices have a small core number, and the number of vertices becomes fewer as the core number grows larger. For the generated graphs, as shown in Fig. 7, all vertices have a core number of 8 in the BA graphs. In the $BTER$ graph, the largest core number of vertices is 20, and most vertices have a small core number. In the RM graph, as the core number k increases the number of vertices with core number k decreases, which is the same as the real-world graphs. As shown later, the core number distribution of a graph will affect the performances of our algorithms.

We use the *average processing time per edge* as the efficiency measurement of the algorithms, such that the efficiency of the algorithms can be compared in different cases. And all the experiments are conducted 20 times to avoid contingency. In the figures given in subsequence, the unit of time in y-axis is microsecond, using μs for short. To check if our

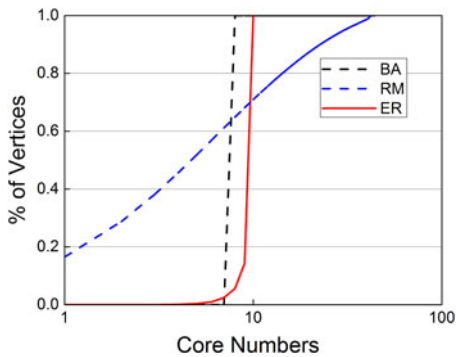
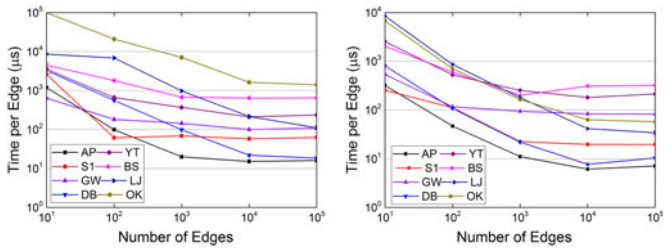
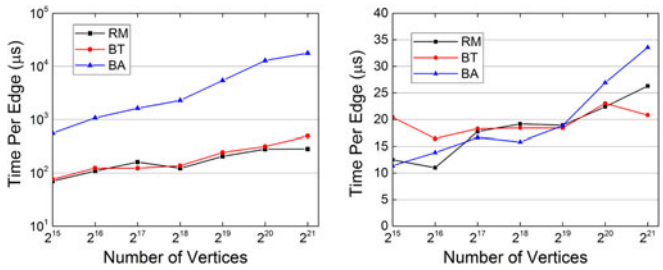


Fig. 7. Core number distribution of generated graphs.



(a) Insertion (b) Deletion

Fig. 8. Impact of the number of edges inserted/deleted in real-word graphs.



(a) Insertion (b) Deletion

Fig. 10. Impact of orginal graph size.

algorithms can get correct results, we recompute the core numbers of all vertices using the static algorithm proposed in [6] when edges are inserted into/deleted from the graph. Then we compare the results computed by our algorithms with the correct results by checking if the core number of each vertex is the same.

In the following sections, we evaluate the impact of two main factors on the algorithm performance: the number of inserted/deleted edges and the size of the original graph. The first factor mainly influences the iterations needed to process the inserted/deleted edges, and the other affects the processing time in each iteration. We use the real-world graphs and temporal networks to conduct the first evaluation, and the synthetic graphs on the second one.

7.1 Stability Evaluation

First we use the real-world graphs and change the number of updated edges. We randomly insert/delete P_i edges from the original graph, where $P_i = 10^i$ for $i = 1, 2, 3, 4, 5$. Figs. 8a and 8b show the processing time per edge for insertion and deletion cases respectively. It can be seen that when more edges are updated, the time needed per edge decreases in general. This is because our algorithms have better performance in the case of large amount of updates. In this case, more edges can be selected into the matching processed in each iteration. Therefore, our algorithms are more suitable for processing large amount of updates. Surely, because the total processing time increases as the number of updated edges increases, there is a break point when it is better to run the static computation algorithm to recompute all the core numbers. We will explain it later in the comparison section.

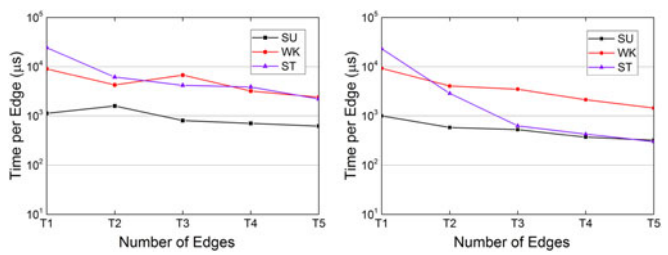
We next test the performance using the temporal networks and vary the number of updated edges as well. Since each edge has a time stamp in the temporal networks, we choose five time points $T1, T2, T3, T4, T5$, where $T1 < T2 < T3 < T4 < T5$ and edges inserted/deleted after $T_i, (1 \leq i \leq 5)$ are regarded as the updated edges. The number of edges after T_i is around 10^{6-i}, which is nearly the same as that in the experiments of real-world graphs. The results are shown in Figs. 9a and 9b. We get similar results as those for the real-word graphs. As the number of updated edges increases, the time needed per edge decreases.

7.2 Scalability Evaluation

We test the scalability of our algorithms using the three synthe tic datasets. The results are shown in Figs. 10a and 10b. With the average degree fixed as 8, we vary the number of vertices $|V|$ from 2^{15} to 2^{21}. For each graph, we randomly select 10000 edges from each graph and show the processing time per edge as the graph size ranges. It can be seen from Figs. 10a and 10b that when the graph size increases at an exponential rate, the processing time increases linearly. This indicates that our algorithms are stable upon different graph size and can handle graphs with extremely large size in acceptable time. As the figure shows, the processing time for BA graph is longer than the other two graphs. This is because in BA graphs all vertices have the same core number 8, the parallelism of our algorithms is not very well in this case, since initially there is only one processor used in the algorithm execution. This situation can also be seen as the worst case for our algorithms. However, real-world graphs do not exhibit this property as shown in Figs. 5 and 6.

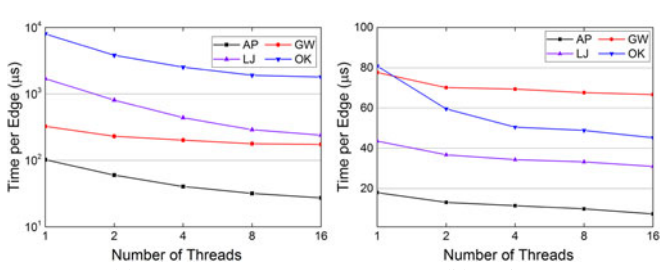
7.3 Parallelism Evaluation

In this section, we show the parallelism of our algorithm. We vary the number of threads to see the speedup of the processing time. As we mentioned before, the parallelism is implemented through a thread pool. We maintain a task queue and a thread queue. The thread queue has a fixed number of threads and the number of tasks is determined by the algorithm and the property of the graph. To exhibit the parallelism, we keep the task queue the same and change the number of threads in the thread pool. The results are shown in Figs. 11a and 11b. As we can see, for the



(a) Insertion (b) Deletion

Fig. 9. Impact of the number of edges inserted/deleted in temporal graphs.



(a) Insertion (b) Deletion

Fig. 11. Vary number of threads for incremental/decremental case.

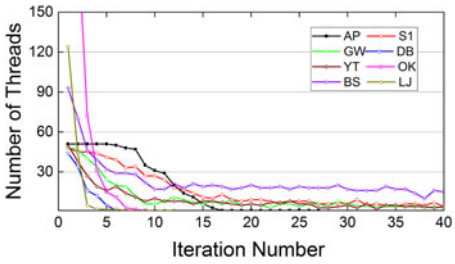


Fig. 12. Number of threads in each iteration when inserting 20K edges.

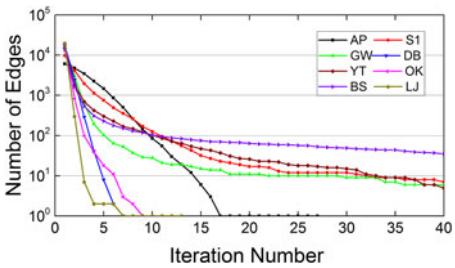
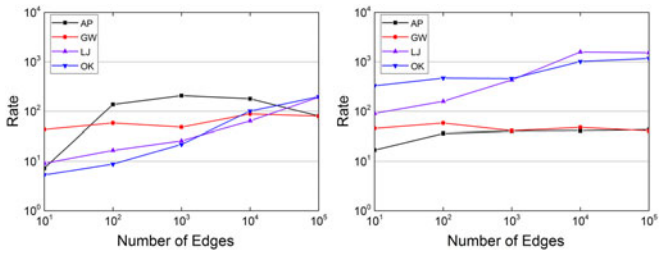
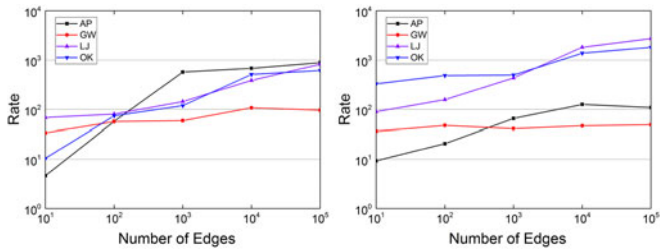


Fig. 13. The number of edges handled in the first 40 iterations.



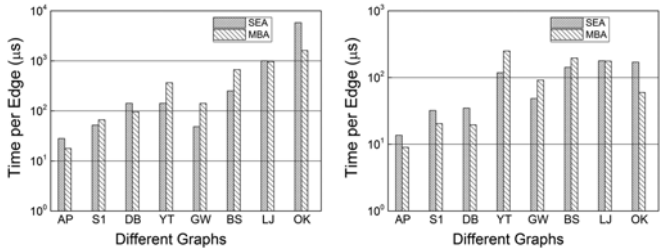
(a) Insertion (b) Deletion

Fig. 14. Comparison of single-thread implementation with the TRAVERSAL Algorithm.



(a) Insertion (b) Deletion

Fig. 15. Comparison of multi-thread implementation with the TRAVERSAL Algorithm.



(a) Insertion (b) Deletion

Fig. 16. Comparison with the Algorithm SEA.

insertion case, the time needed per edge decreases when the number of threads increases, especially in large graphs. While in the deletion case, the trend is not as significant as that in the insertion case. This is because the processing time in the deletion case has already been very low. And most time is taken on maintaining the thread pool, while the computation time of the algorithm only occupies a small part. So when we change the number of threads, the total processing time does not vary very much.

The number of threads needed and the number of edges processed in each iteration in the first 40 ones are illustrated in Figs. 12 and 13, respectively, when inserting 20K edges with a fixed thread number 16 in the thread pool. In our algorithm, the number of threads needed is the number of tasks in the task queue. The figures show that at the beginning, the threads are fully made use of, and large amount of edges are processed in parallel. As the number of updated edges left decreases, the parallelism becomes poorer.

7.4 Comparisons with Other Algorithms

In this section, we compare our matching based algorithms (MBA) with other algorithms, including the state-of-the-art sequential algorithms, *TRAVERSAL* algorithms (Algorithms 5 and 7) given in [24], the parallel superior edge algorithms (SEA) proposed in [28]. We also compare the proposed MBA algorithms with the core decomposition algorithm for static graphs in [6], to check the break point for recomputing the core numbers.

The comparison results with the *TRAVERSAL* algorithms are illustrated in Figs. 14a, 14b, 15a and 15b. The comparisons are conducted on four typical real-world graphs, AP, GW, LJ and OK as given in Table 1. For each graph, we randomly select {10, 100, 1000, 10000, 100000} edges as the update set. In the figures, the y-axis represents the speedup rate. As mentioned before, the speedup of our algorithms comes from two optimizations: (i) the reduction of unnecessarily repeated visit of edges and vertices; and (ii) parallel processing. Hence, we implement our algorithms using a single thread and multiple threads respectively, to test the impacts of the two optimizations respectively. Figs. 14a and 14b show the speedup rate of single-thread implementations of our algorithms over the *TRAVERSAL* algorithms, and Figs. 15a and 15b show the comparisons of multiple-thread implementations of our algorithms and the *TRAVERSAL* algorithms. As shown in the figures, both optimizations contribute to the speedup. Furthermore, it can be found that by reducing duplicated visits of edges and vertices, the processing time are greatly decreased. It can be also found that the speedup rate increases as the number of edges inserted/deleted increases. This is because when there are more edges inserted/deleted, more edges can be selected into a matching and processed together, which can reduce more duplicated visits of edges and vertices in sequential algorithms. Hence, our algorithms provide better efficiency in large-scale graphs and processing large amount of graph changes.

Figs. 16a and 16b show the comparison of our algorithms and the superior edge based algorithms given in [28] on the processing time per edge. In principle, our algorithms need less preprocessing time, while the superior edge based algorithms may select more edges processed together. From the figures, it can be seen that in the insertion case, our algorithm exhibits better performance in 4 of 8 graphs, and in the deletion case, our algorithm perform better in

TABLE 3
Comparison with Static Algorithm

Datasets	Break Point
AP(ca-Astroph)	2.35K
S1(soc-Slashdot)	2.69K
DB(DBLP)	20.7K
YT(YouTube)	2.959K
GW(Gowalla)	1.3K
BS(web-BerkStan)	1.89K
LJ(LiveJournal)	137.7K
OK(Orkut)	283.75K

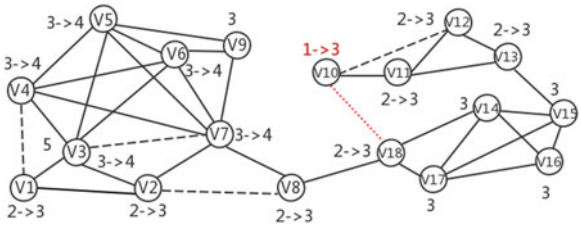


Fig. 17. A matching $\{<v_1, v_4>, <v_3, v_7>, <v_2, v_8>, <v_{10}, v_{12}>\}$ and an extra edge $<v_{10}, v_{18}>$ are inserted. The core number of vertex v_{10} will increase by 2, from 1 to 3.

5 of 8 graphs. Specially, in two large graphs LJ and OK, our algorithms are faster.

Table 3 illustrates the break points for our algorithms, i.e., the number of edges whose processing time breaks the time needed for recomputing the core numbers, using the static core decomposition algorithm in [6]. As shown in the table, the break point can be as large as hundreds of thousands of edges in large graphs. Because the size of real data can be much more larger than the data sets in our experiments. Hence, our algorithms can performance well in reality.

Evaluation Summary. The experiment results show that our algorithms exhibit good stability and scalability. Comparing with sequential algorithms, our algorithms speed up the core maintenance procedure significantly. Additionally, our algorithms are suitable for handling large amount of edge insertions/deletions in large-scale graphs, which is desirable in realistic implementations.

8 Conclusion

In this paper, we present parallel algorithms for core maintenance in dynamic algorithms. Our algorithms exhibit significant acceleration comparing with sequential processing algorithms that handle inserted/deleted edges sequentially. Experiments on real-world and synthetic graphs illustrate that our algorithms implement well in reality, especially in scenarios of large-scale graphs and large amounts of edge insertions/deletions.

Our algorithms provide a general framework for handling updated edges in parallel in the core maintenance process, i.e., finding an available set of edges containing as many edges as possible that makes the core number of every vertex change by at most 1. We have shown that matching is an approach fulfilling this requirement, and it may not work anymore if we add an arbitrary edge to the matching, as shown in Fig. 17. Hence, a meaningful direction for future work is to derive a structure that contains more edges and fits our framework, which will help get algorithms with better parallelism.

Acknowledgments

This work is supported by the National Key R&D Program of China 2018YFB1003203, the National Natural Science Foundation of China Grants 61602195, 61572216, 61433019 and U1435217, Natural Science Foundation of Hubei Province 2017CFB301, the Outstanding Youth Foundation of Hubei Province 2016CFA032, and Fundamental Research Funds for HUST.

References

[1] H. Aksu, M. Canim, Y. C. Chang, I. Korpeoglu, and Ö. Ulusoy, "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. 18th Int. Conf. Neural Inf. Process. Syst.*, 2005, 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," in *BMC Bioinformatics*, 2010, Art. no. 2.

[5] A.L. Barabasi and R. Albert, "Emergence of scaling in random networks," in *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.

[7] F. Bonchi, F. Gullo, A. Kaltenbrunner, and Y. Volkovich, "Core decomposition of uncertain graphs," in *Proc. 20th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2014, pp. 1316–1325.

[8] S. Carmi, S. Havlin, S. Kirkpatrick, Y. Shavitt, and E. Shir, "From the cover: A model of internet topology using K-shell decomposition," *Proc. National Acad. Sci.*, 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. 4th 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] C. Demetrescu, I. Finocchi, and G. F Italiano, *Handbook of Data Structures and Applications*. Boca Raton, FL, USA: CRC Press, 2004.

[13] T. G. Kolda, A. Pinar, and T. Plantenga, et al., "A scalable generative graph model with community structure," *SIAM J. Sci. Comput.*, vol. 36, no. 5, 2013.

[14] R. A. Hanneman and M. Riddle, *Introduction to Social Network Methods*. Riverside, CA, USA: Univ. California Riverside, 2005.

[15] T. Hayashi, T. Akiba, and Y. Yoshida, "Fully dynamic betweenness centrality maintenance on massive networks," in *Proc. VLDB Endowment*, vol. 9, no. 2, pp. 48–59, 2015.

[16] L. Jure and K. Andrej, SNAP Datasets. Stanford Large Network Dataset Collection, 2014. [Online]. Available: http://snap.stanford.edu/data.

[17] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo, "*K*-core decomposition of large networks on a single PC," in *Proc. VLDB Endowment*, vol. 9, no. 1, pp. 13–23, 2016.

[18] M. Kitsak, L. K. Gallos, S. Havlin, and H. A. Makse, "Identification of influential spreaders in complex networks," in *Nature Phys.*, vol. 6, no. 11, pp. 888–893, 2010.

[19] 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, 2014.

[20] P. Meyer, H. Siy, and S. Bhowmick, "Identifying Important Classes of Large Software Systems through *K*-core Decomposition," *Adv. Complex Syst.*, 2015.

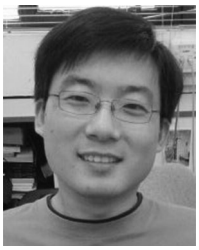
[21] J. Misra and D. Gries, "A constructive proof of Vizing's Theorem," *Inf. Process. Lett.*, vol. 41, no. 3, pp. 131–133, 1992.

[22] 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.

[23] M. P. O'Brien and B. D. Sullivan, "Locally Estimating Core Numbers," in *Proc. Int. Conf. Data Mining*, 2014, pp. 460–469.
[24] A.E. Sariyüce, B. Gedik, G. Jacques-Silva, K. Wu, and Ü. V. Catalyürek, "Incremental *K*-core decomposition: Algorithms and evaluation," *VLDB J.*, vol. 25, no. 3, pp. 425–447, 2016.
[25] A.E. Sariyuce, C. Seshadhri, A. Pinar, and Ü. V. Catalyürek, "Finding the Hierarchy of Dense Subgraphs using Nucleus Decompositions," in *Proc. 24th Int. Conf. World Wide Web*, 2015, pp. 927–937.
[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.
[27] H. Wu, J. Cheng, Y. Lu, Y. Ke, Y. Huang, D. Yan, and H. Wu, "Core decomposition in large temporal graphs," in *Proc. IEEE Int. Conf. Big Data*, 2015, pp. 649–658.
[28] 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.
[29] T. Zhang, X. Wang, Z. Li, F. Guo, Y. Ma, and W. Chen, "A survey of network anomaly visualization," *Science China Inf. Sci.*, 10.1007/s11432-016-0428-2, pp. 121101:1–121101:17, 2017.
[30] Y. K. Zhang, J. X. Yu, and Y. Zhang, "A fast order-based approach for core maintenance," in *Proc. IEEE 33rd Int. Conf. Data Eng.*, 2017, pp. 337–348.

Hai Jin received the PhD degree from 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 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. He is a senior member of the IEEE.

Na Wang received the BE degree from Computer School, JiLin University, in 2015. She is currently working toward the master's degree in the Department of Computer Science, Huazhong University of Science and Technology (HUST). Her research interests include dynamic graphs, social networks and parallel computing.



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

Qiang-Sheng Hua received the BEng and MEng degrees from the School of Information 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 an associate professor with Huazhong University of Science and Technology, China. He is interested in parallel and distributed computing, including algorithms and implementations in real systems. He is a member of the IEEE.

Xuanhua Shi received the PhD degree in computer engineering from Huazhong University of Science and Technology (China), in 2005. He is a professor with Big Data Technology and System Lab/Service Computing Technology and System Lab, Huazhong University of Science and Technology (China). From 2006, he worked as an INRIA post-doc in PARIS team with Rennes for one year. His current research interests include cloud computing and big data processing. He published more than 90 peer-reviewed publications, received research support from a variety of governmental and industrial organizations, such as National Science Foundation of China, Ministry of Science and Technology, Ministry of Education, European Union and so on. He has chaired several conferences and workshops, and served on technical program committees of numerous international conferences. He is a senior member of the IEEE and CCF, and a member of the ACM.

Xia Xie received the PhD degree in computer architecture from Huazhong University of Science and Technology, in 2006. She is an associate professor with Huazhong University of Science and Technology, in China. Her research interests include performance evaluation and data mining.