Core Maintenance on Dynamic Graphs: A Distributed Approach Built on H-Index

Qiang-Sheng Hua, *Member, IEEE*, Hongen Wang, Hai Jin, *Fellow, IEEE*, and Xuanhua Shi, *Senior Member, IEEE*

***Abstract*—Core number is an essential tool for analyzing graph structure. Graphs in the real world are typically large and dynamic, requiring the development of distributed algorithms to refrain from expensive I/O operations and the maintenance algorithms to address dynamism. Core maintenance updates the core number of each vertex upon the insertion/deletion of vertices/edges. Although the state-of-the-art distributed maintenance algorithm (Weng et al.˜2022) can handle multiple edge insertions/deletions simultaneously, it still has two aspects to improve. (I) Parallel processing is not allowed when inserting/removing edges with the same core number, reducing the degree of parallelism and raising the number of rounds. (II) During the implementation phase, only one thread is assigned to the vertices with the same core number, leading to the inability to fully utilize the distributed computing power. Furthermore, the h-index (Lü, et al. 2016) based distributed core decomposition algorithm (Montresor et al. 2013) can fully utilize the distributed computing power where all vertices can be processed in parallel. However, it requires all vertices to recompute their core numbers upon graph changes. In this article, we propose a distributed core maintenance algorithm based on h-index, which circumvents the issues of algorithm (Weng et al.˜2022). In addition, our algorithm avoids core numbers recalculation where the numbers do not change. In comparison to the state-of-the-art distributed maintenance algorithm (Weng et al.˜2022), the time speedup ratio is at least 100 in the scenarios of both insertion and deletion. Compared to the distributed core decomposition algorithm (Montresor et al. 2013), the average time speedup ratios are 2 and 8 for the cases of insertion and deletion, respectively.**

***Index Terms*—Core maintenance, distributed algorithm, dynamic graphs, h-index.**

I. Introduction

K-CORE is a crucial instrument for analyzing the cohesion of graphs. The k-core of a graph is a maximal induced subgraph in which each vertex has at least k neighbors. Core number of vertex v equals the maximum k in all k-core subgraphs containing vertex v. For an edge $e = (u, v)$, the core number of e equals the minimum of u and v's core numbers.

Manuscript received 19 February 2023; revised 16 November 2023; accepted 25 December 2023. Date of publication 11 January 2024; date of current version 4 September 2024. This work was supported in part by the National Key Research and Development Program of China under Grant 2022ZD0115301, and in part by the National Natural Science Foundation of China under Grant 61832006. Recommended for acceptance by N. Cao. *(Corresponding author: Qiang-Sheng Hua.)*

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

Digital Object Identifier 10.1109/TBDATA.2024.3352973

Numerous applications, such as detecting community in temporal networks [18], [19], finding impactful spreaders in highly complicated networks [20], [21], exploring the collective emotions in social networks [26], analyzing large-scale software systems [24], [25], anticipating the features of biology networks [23], and visualizing big networks [22], have made extensive use of k-core.

As a result of the popularity of networking, real graph applications exhibit the following characteristics: (1) some graphs might be too large to fit a single machine's memory; (2) most graph applications are dynamic, meaning the insertion/deletion of vertices/edges is frequent. For the memory issue, we can turn to distributed algorithms; for the dynamism of graphs, we typically employ maintenance techniques which try not to recompute all vertices' core numbers upon graph changes.

The previous algorithms for core maintenance can be categorized as follows: (I) the centralized sequential algorithms [2], [3] process a single edge at a time; (II) the centralized parallel algorithms [5], [4], [6] can simultaneously process several edges; (III) the distributed algorithms [8] process one edge at a time; (IV) the distributed algorithm [9] can simultaneously process multiple edges. It is important to note that (I) and (III) share similar ideas, as do (II) and (IV), but (III) and (IV) are specifically tailored for optimization in a distributed environment.

It is obvious that the parallelism of single-edge processing algorithms is low, thus we will focus on the issues of distributed multi-edge processing algorithm [9]: (I) when the inserted/deleted edges have the same core number, they need to be tackled sequentially; (II) in the implementation phase, only one thread is assigned to the vertices with the same core number, leading to the waste of distributed computing power since there are only a few core numbers for a large graph. For example, the graph KONECT[1], a social friendship network consisting of people and their friendship ties, has 59.2 million vertices from multiple connected components, but the maximum core number is 17, which means that when we have more than 17 processors, the computing power cannot be fully utilized.

Due to the above two issues, the distributed core maintenance algorithm [9] has a relatively low parallelism. Meanwhile, the distributed core decomposition algorithm [10] based on h-index [1] could be highly parallel since all vertices can work simultaneously. H-index reflects the connection between

[1]https://networkrepository.com/socfb-konect.php

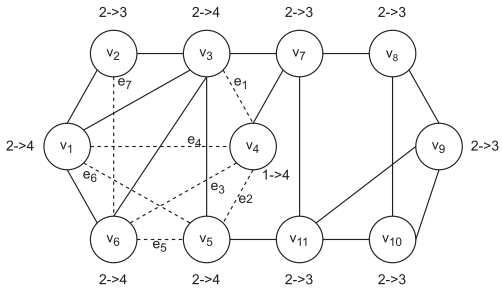


Fig. 1. Dashed lines represent the inserted edges. The number atop each vertex represents the alteration of the vertex's core number.

degree and core number, which is applied in distributed core decomposition [10] and shared-memory core maintenance [28].

The distributed core decomposition algorithm consists of two steps: (I) initialization: each vertex's h-index is set to its degree; (II) convergence: each vertex updates its h-index based on its neighbors' h-indices. Convergence ends when all vertices' h-indices no longer change, and now each vertex's h-index equals its core number.

Now, a natural question is how well the distributed core decomposition algorithm performs on the core maintenance of dynamic graphs. Surprisingly, according to Figs. 10 and 11 in Section VI, we found that when there are a batch of inserted/deleted edges, directly applying the distributed core decomposition algorithm may have a much better performance than the distributed core maintenance algorithm. The following examples will give the readers a concrete feeling of the above discussions.

Core Maintenance [9]: (I) the algorithm requires that edges with the same core number cannot be processed simultaneously. Take Fig. 1 as an example, in the beginning stages, the core numbers of $e_1, \ldots, e_4$ are 1, and the core numbers of e_5, e_6, e_7 are 2; therefore, only two edges (one from $\{e_1, \ldots, e_4\}$ and the other from $\{e_5, e_6, e_7\}$) can be tackled simultaneously. In addition, when there are a significant number of inserted/deleted edges, the preprocessing will also incur non-negligible overhead; (II) also in Fig. 1, when we insert e_1, the core number of v_4 changes to 2. Then if we insert e_2, the core numbers of $v_3, v_4, v_5, v_7, \ldots, v_{11}$ will change from 2 to 3. We can see that $v_3, v_4, v_5, v_7, \ldots, v_{11}$ have the same core number. Therefore, the updates of these vertices can only be sequential.

Core Decomposition [10]: after $e_1, \ldots, e_7$ have been added, by using the distributed decomposition algorithm, we only need one round to recompute the core numbers. Furthermore, if we have 11 processors (the graph in Fig. 1 has 11 vertices), all vertices can work in parallel. Note that, if we only insert one edge e_1, the distributed core decomposition will take four rounds. The results can be verified in the same way as Table II in Section V-B. The reason for the above anomaly is due to the folowing fact: the algorithm initializes its h-index as its degree, when only e_1 is inserted, the vertices have larger differences between their core numbers and degrees. The lower the initial h-indices are, the faster the algorithm converges.

In summary, the distributed core maintenance algorithm [9] suffers from limited parallelism, while the distributed core decomposition algorithm [10] involves redundant re-computation of core numbers for unchanged vertices, resulting in a higher initial h-index. Thus, there is a need to devise a more efficient distributed core maintenance algorithm that enhances parallelism and minimizes the initial h-index and re-computation of core numbers for unchanged vertices.

In this paper, we propose a method for incorporating the h-index [1] into core maintenance by adjusting its initial value. For edge insertion in the graph, we leverage the insights from previous algorithms [2][4] and derive an upper bound of the core numbers. This upper bound is then used to initialize the h-index. The theoretical perspective of this upper bound ensures that the initial h-index is kept as low as possible, as it represents the maximum core number achievable for each vertex.

Regarding edge deletion from the graph, our analysis demonstrates that the core number before the graph update acts as the upper bound in this scenario. This approach ensures that the initial h-index is appropriately set, minimizing computational overhead and improving the efficiency of the core maintenance algorithm.

The following summarizes our contributions:

- We present thorough theoretical analyses on core numbers change to get lower upper bounds of initial h-index under various scenarios.
- We devise h-index based distributed core maintenance algorithms that can handle multiple edges with the same core number. Both a global updating technique and a pipelined traversal-based technique are designed to obtain a lower initial h-index.
- Extensive experimental results on real-world graphs provide evidence that our distributed algorithms are stable, scalable, highly parallel, and efficient.

The remainder of the article is structured as follows: Section II provides some key definitions. We introduce the system model and the distributed graph processing systems in Section III. In Section IV, we will introduce the asynchronous h-index algorithm. The distributed core maintenance algorithm for insertion and deletion will be introduced in Section V. The findings of our comprehensive experiments are reported in Section VI. In Section VII, we discuss related work. Section VIII concludes the paper.

II. Preliminaries

We focus on an unweighted and undirected graph $G = (V(G), E(G))$, where $V(G)$ represents the set of vertices and $E(G)$ represents the set of edges in G. $d_G(v)$ represents vertex v's degree in G, and $neb(v)$ represents the set of neighbors of vertex v in G. $\delta(G)$ represents the minimum vertex degree in G. We say a graph H is a subgraph of graph G, if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$.

Definition 1 (k-core). A k-core of G is a maximal induced subgraph H where $\delta(H) \geq k$.

Definition 2 (Vertex-core). If v is included in a k-core and no $(k+1)$-cores include v, v's core number equals k. We call it $C_G(v)$.

TABLE I
SUMMARY OF NOTATIONS

Notations	Description
$V(G)$	the set of vertices in G
$E(G)$	the set of edges in G
$d_G(v)$	The vertex v' degree in G
$neb(v)$	The set of neighbors of vertex v
$\delta(G)$	The minimum vertex degree of graph G
$C_G(v)$	Vertex v's core number in G
$C_G(e)$	Edge e's core number in G
$sd(v)$	Vertex v's superior degree
$nise(v)$	The number of inserted superior edges of vertex v
$h^i(v)$	Vertex v's h-index at round i
$G(S)$	The induced subgraph of the edge set S
$V(S)$	The vertex set of edges set S
$G_I(S)$	Graph $G_I(S) = G + G(S)$
$G_D(S)$	Graph $G_D(S) = G - G(S)$

Definition 3 (Edge-core). For an edge $e = (u, v)$ in G, we call its core number as $C_G(e) = min\{C_G(u), C_G(v)\}$.

Definition 4 (Superior Edge [4]). For an edge $e = (u, v)$, if $C_G(u) \leq C_G(v)$, we say edge e is a superior edge of vertex u.

Definition 5 (Superior Degree [6]). The superior degree of a vertex v in G is equal to the number of its superior edges, denoted as $sd(v)$.

Definition 6 (Number of Inserted Superior Edges). Suppose S is the inserted edges set. For an edge $e = (u, v)$, $e \in S$, if e is a superior edge of vertex v, then e is an inserted superior edge of v. For a vertex v, we call its number of inserted superior edges as $nise(v)$. Obviously, if v is not in $V(S)$ (the set of all vertices in S), $nise(v) = 0$.

Definition 7 (h-index [1]). The h-index reflects the relationship between the vertex degree and the core number. It is initialized with the degree of a vertex, and finally converges to the core number. The h-index is defined as the largest integer h for which the vertex has at least h neighbors whose h-indices are greater than or equal to h. For a vertex v in G, vertex v's h-index at round i is called $h^i(v)$. From [1], we can infer that if $h^0(v) \geq C_G(v)$, then $h^\infty(v) = C_G(v)$. When the number of rounds is not specified, denote $h^i(v)$ as $h(v)$.

Definition 8 (Superior edges set [4]). Suppose S is the inserted/deleted edges set. If vertex $v \in V(S)$, and v only has at most one superior edge in S, we call S a superior edges set.

Definition 9 (Reachable Tree). Suppose S is the inserted/deleted edges set to G, $e = (u, v)$ is an edge in the set S, and $C_G(u) \leq C_G(v)$. The reachable tree of vertex u demands that the core number of each vertex on the tree equals $C_G(u)$. In addition, for each vertex x in the reachable tree, there exists at least one path in the tree connecting x to u.

III. SYSTEM OVERVIEW

A. System Model

The pregel-like [11] distributed graph processing system model is adopted where there are n fully connected machines and each machine holds a subgraph of the input graph G with N vertices. The calculation is based on the Bulk Synchronous Parallel(BSP) model [31], which is executed asynchronously within the machine and synchronously between the machines. Each machine is allowed to perform the following operations: (1) for a vertex u in G locating on machine i, suppose v is a neighbor of u and v is located on machine j, machine i can send the core number of u to machine j; (2) each machine is able to save the core numbers sent by other machines in the last round.

B. Distributed Graph Processing Systems

Numerous pregel-like distributed graph processing systems exist, including Power-Graph [15], GraphX [14], PowerLyra [13], Gemini [12], and D-Galois [16]. D-Galois and Gemini have the best performance among them [12][16]. These graph processing systems all fit our system model. Since Gemini has a smaller code size and an easier-to-use interface than D-Galois, we choose it to implement our distributed algorithms. Following is a brief introduction to Gemini [12].

1) Graph Partition:

- **Vertices partition**: they are evenly distributed to each machine based on the vertex's id. Suppose there are p machines, vertex v_i is placed on machine $\lfloor \frac{i}{N/p} \rfloor$.
- **Edges partition**: if vertex u is located on machine i, and u is an endpoint of the undirected edge e, then e is placed on machine i.

2) Vertex and Edge Functions: Gemini [12] offers the vertex and edge functions to implement the distributed algorithm. The $work\_list$ is a set of vertices that need to be computed.

- **Vertex Function**: the function performs vertex programs for the vertices in the $work\_list$. If vertex v is located on machine i, then machine i will execute the vertex program of v. For a vertex v, we can access the most recent data of its neighbors on the same machine using Gemini.
- **Edge Function**: the function performs edge programs for edges of vertices in the $work\_list$. Gemini's edge function is separated into sparse and dense modes. Sparse mode edge function broadcasts the messages of the vertices in the $work\_list$ before executing programs, whereas dense mode executes the program first and then aggregates the message. Therefore, for a vertex v, we must employ sparse mode edge function to guarantee the vertex to receive all of v's neighbors' raw messages.

C. Round and Message Complexities

We evaluate the distributed algorithm with round and message complexities. The round complexity means the number of BSP super-steps. The message complexity is the total number of messages delivered in the system. Each message in our system consists of $O(logN)$ bits, where N represents the number of vertices.

IV. DISTRIBUTED ASYNCHRONOUS H-INDEX ALGORITHM: THE SUBROUTINE OF CORE MAINTENANCE

In this Section, we will present the asynchronous H-Index algorithm, which serves as a fundamental component of the core maintenance algorithm. The subsequent process of the core maintenance algorithm invokes this algorithm to determining the upper bound of the h-index.

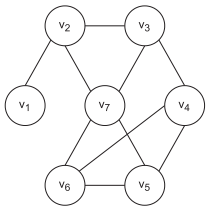


Fig. 2. An illustrating graph showing the difference between the asynchronous H-Index algorithm and core decomposition algorithm.

The distributed core decomposition algorithms [10] are also built on the h-index, but it assumes that all vertices' h-indices are updated synchronously. However, the number of machines is far smaller than the number of vertices in the graph, indicating that a portion of the processing of vertices is sequential. For a vertex v, we can access the most recent data of its neighbors on the same machine using Gemini. We propose an asynchronous H-Index algorithm based on this observation. The difference between the asynchronous and the synchronous algorithm is that the latter calculates using data from the last round. The asynchronous algorithm, in contrast, utilizes the most recent data in the current round. Consequently, the asynchronous algorithm can converge more rapidly.

Let $h(v)$ be the h-index of vertex v during the convergence process. We have the following (1) where $neb(v)$ represents the set of neighbors of vertex v. In each round, each vertex in the $work\_list$ will update its h-index based on this equation.

$$h(v) = \max_{k \geq 0} \{k : | \{u \in neb(v) : h(u) \geq k\} | \geq k\} \quad (1)$$

The detailed distributed asynchronous H-Index algorithm is listed in Algorithm 1. Each machine is allocated an array named h of size $|V(G)|$. The h array is initialized as $h^0(v)$ of all vertices in G, and the $work\_list$ is the set of vertices with h-indices greater than or equal to their core numbers. In the distributed core decomposition algorithm, $h^0(v)$ is initialized as $d_G(v)$, but we only require $h^0(v) \geq C_G(v)$. In line 5, Each vertex updates its h-index according to (1) (Algorithm 2). The vertex is added to the $send\_list$ when its h-index changes (lines 6-8). For a vertex v, $send\_list.set(v)$ means adding v into $send\_list$. If vertex u and vertex v are located on the same machine, and u executes the vertex program first, then v will use the latest $h(u)$ when implementing the vertex program.

After all vertices' h-indices have been updated, the algorithm synchronizes the h-indices of the vertices in the $send\_list$ to all machines (line 12). For a vertex v in $send\_list$, only neighbors with larger h-indices than $h(v)$ are possible to change, thus the algorithm adds these neighbors to the new $work\_list$ (lines 13-15). When all the vertices' h-indices remain unchanged, it means that $send\_list$ is empty; thus we can deduce new $work\_list$ is empty which means the algorithm terminates.

We will give an example to demonstrate the difference between the asynchronous H-Index algorithm and distributed core decomposition algorithm. For the graph in Fig. 2, as shown in Table II, the synchronous distributed core decomposition algorithm requires three rounds. Assuming the execution order in the

Algorithm 1: Async-H-Index(G, h, $work\_list$).

```
   Input  : G = (V(G), E(G)), h^0, work_list
   Output: all vertices' core numbers in G
1  while work_list is not empty do
2      send_list ← empty;
3      for v ∈ work_list do
4          //Gemini's vertex function
5          t ← computeH(neb(v), h(v));
6          if t < h(v) then
7              h(v) ← t;
8              send_list.set(v);
9      work_list ← empty;
10     for u ∈ send_list do
11         //Gemini's edge function
12         sync h(u);
13         for v ∈ neb(u) do
14             if h(u) < h(v) then
15                 work_list.set(v);
16 return h;
```

Algorithm 2: computeH($neb(v), h(v)$).

```
   Input  : The neighbors of vertex v, h-index of v
   Output: updated h-index of v
1  for u ∈ neb(v) do
2      if h(u) > h(v) then
3          count[h(v)] ← count[h(v)] + 1;
4      else
5          count[h(u)] ← count[h(u)] + 1;
6  for i ← h(v) down to 1 do
7      count[i − 1] ← count[i − 1] + count[i];
8  i ← h(v);
9  while i > 1 and count[i] < i do
10     i ← i − 1;
11 return i;
```

TABLE II
THE EXECUTION OF DISTRIBUTED CORE DECOMPOSITION FOR THE GRAPH IN FIG. 2

Round	$h(v_1)$	$h(v_2)$	$h(v_3)$	$h(v_4)$	$h(v_5)$	$h(v_6)$	$h(v_7)$
0	1	3	3	3	3	3	4
1	1	2	3	3	3	3	3
2	1	2	2	2	3	3	2
3	1	2	2	2	2	2	2

H-Index algorithm is v_1, v_2,..., v_7. In this instance, the vertex program is executed only once per vertex to get the final result, so the asynchronous H-Index algorithm needs one round. The worst-case scenario for the asynchronous H-Index algorithm is that the neighbors' h-indices have not been updated in the same round of calculation. Hence, the asynchronous H-Index algorithm will use the last round's h-indices. For this case, the asynchronous H-Index will degenerate into the synchronous version.

V. Distributed Core Maintenance

This section will introduce our distributed core maintenance algorithms, including the incremental core maintenance and decremental core maintenance algorithms.

A. Incremental Core Maintenance

The state-of-the-art distributed core maintenance algorithm [9] cannot handle edges with the same core number simultaneously. Our core maintenance algorithms do not have the above limitation and they are based on Algorithm 1. We add all vertices whose core numbers may change to the $work\_list$, attempting to determine the initial h-indices of these vertices.

As mentioned before, the challenges are to make the initial h-indices as low as possible and to identify the vertices whose core numbers may change. Suppose the inserted edges set is S. For a vertex v in $V(G_I(S))$ where $G_I(S)$ is the new graph after inserting S, if $h^0(v) = C_G(v)$ is true, it means that the vertex v's core number remains unchanged. Otherwise v needs to be added to the $work\_list$.

In the insertion case, it is apparent that a candidate for the initial h-index is the vertex's degree. However, this trivial solution makes the core maintenance algorithm the same as the core decomposition one. We desire the initial h-indices to be as tight as feasible, ideally equal to the vertex's updated core number. A naive idea is to update the initial h-indices based on the set of candidates in the TRAVERSAL algorithm [2]. This could be a viable solution when the insertion set has only one edge. However, when the insertion set contains multiple edges, we can only insert these edges sequentially to obtain the correct candidate set, which diminishes parallelism. An efficient algorithm to calculate initial h-indices is needed to overcome the issue.

Before introducing our algorithm, we need to present some theoretical findings about the initial h-indices.

1) Theoretical Basis: The following is the characteristic of the superior edge set that will be used in the subsequent analysis.

Lemma 1 ([4]). Given a graph $G = (V(G), E(G))$, when a superior edges set S is inserted into graph G, and each edge in S has the same core number α. the change of each vertex's core number is at most one.

Now we know the change in vertices' core numbers when the superior edge set S with core number α is inserted into the graph G. Then we will analyze the change in vertices' core numbers when an ordinary edge set with core number α is inserted into the graph G.

Lemma 2. Suppose S is the inserted edges set, and each edge in S has the same core number α. Denote $V_\alpha = \{v : C_G(v) = \alpha, v \in V(S)\}$, $m = \max_{v \in V(S)} nise(v)$. There exists a set $S_\alpha \subseteq S$ satisfying the following conditions. After S_α is inserted into G, each vertex's core number changes at most one. If $S - S_\alpha$ is not empty, then we have $C_{G_I(S_\alpha)}(e) = \alpha + 1$, $e \in S - S_\alpha$, and $\max_{v \in V(S - S_\alpha)} nise(v) \leq m - 1$.

Proof. First, we select a superior edges set SE_1 from S where SE_1 satisfies the following two conditions: (I) each vertex in V_α needs to select a superior edge; (II) if vertex $v \in V_\alpha$, and v has multiple superior edges, then only the one with the lowest core number can be selected. From Lemma 1, we know that after SE_1 is inserted, the change of each vertex's core number is at most one.

Now the vertices in V_α can be divided into two categories: (1) the vertices whose core numbers increased by one; (2) the vertices with unchanged core numbers. For vertices of category (2), we select SE_2 from $S - SE_1$ in a similar way as selecting SE_1 from S. If $SE_2 + SE_1$ is inserted into G, only the core number of the vertex whose core value is α may be changed by at most one. We repeat this process until the core number of each vertex v ($v \in V_\alpha$) has been changed, or v does not have superior edges. We call the selected edges set S_α. Since each vertex selected at least one superior edge, thus $\max_{v \in S - S_\alpha} nise(v) \leq m - 1$. If $S - S_\alpha$ is not empty, for edge $e = (u, v) \in S - S_\alpha$, $C_{G_I(S_\alpha)}(e) = \min(C_{G_I(S_\alpha)}(u), C_{G_I(S_\alpha)}(v)) = \alpha + 1$.

Lemma 2 analyzes the partition of the edge set after inserting the edge set S with the core number α into the graph G. This conclusion will be utilized later to establish the upper bound of the vertices' core numbers. As the calculation of the upper bound aims to initialize the h-index and the initial value is denoted as $h^0(v)$.

Lemma 3. Suppose S is the inserted edges set, and each edge has the same core number α. Denote $m = \max_{v \in V(S)} nise(v)$. For a vertex v in graph $G_I(S)$, we have:

$$h^0(v) = \begin{cases} C_G(v), & C_G(v) < \alpha \\ \alpha + m, & \alpha \leq C_G(v) < \alpha + m \\ C_G(v), & C_G(v) \geq \alpha + m. \end{cases} \tag{2}$$

Proof. From Lemma 2, we know that we can select no more than m times of the set like S_α until S is empty. Suppose $S = \{S_\alpha, S_{\alpha+1}, \ldots, S_{\alpha+m-1}\}$. First, when we insert S_α into G, only vertices whose core numbers equal α may be increased by one. We call the set of vertices whose core numbers may increase at this time as V_α. When $S_{\alpha+1}$ is inserted, the vertices whose core numbers equal $\alpha + 1$ may be increased by one. Thus the core number of vertex v in V_α may be increased twice. We can further deduce that when $S_{\alpha+m-1}$ is inserted, if $C_G(v) = \alpha + i$ ($0 \leq i < m$), v's core number may be increased by $m - i$ times. Thus if $\alpha \leq C_G(v) < \alpha + m$, $h^0(v) = \alpha + i + m - i = \alpha + m$. In other cases, the core numbers remain unchanged, so the $h^0(v)$ equals $C_G(v)$.

Lemma 3 analyzes the upper bound of the vertices' core numbers after inserting the edge set S with core number α into graph G. However, in practical applications, the inserted edge set often contains edges with different core numbers. Therefore, the following analysis will provide an upper bound of the vertices' core numbers after an unrestricted edge set is inserted into graph G.

Lemma 4. Suppose S is the inserted edges set, $x = \min_{e \in S} C_G(e)$, and $y = \max_{e \in S} C_G(e)$. We divide S according to the core numbers of edges where $S = \{S_x, S_{x+1}, \ldots, S_{y-1}, S_y\}$. Denote $m_i = \max_{v \in V(S_{x+i})} nise(v)$ $(0 \leq i \leq y - x)$, and $t_j = \max_{0 \leq i \leq j}(m_i - j + i)$ $(0 \leq j \leq y - x)$. For a vertex v in

graph $G_I(S)$, we have:

$$h^0(v) = \begin{cases} C_G(v), & C_G(v) < x \\ C_G(v) + t_{C_G(v)-x}, & x \le C_G(v) < y + t_{y-x} \\ C_G(v), & C_G(v) \ge y + t_{y-x}. \end{cases} \tag{3}$$

Proof. We first insert S_y into G. From Lemma 3, we know the core numbers of vertices whose core values satisfy $y \le C_G(v) < y + m_{y-x}$ may be changed at most to $y + m_{y-x}$. Then, when we insert S_{y-1}, if vertex v's core number changes, we conclude $y - 1 \le C_G(v) < y - 1 + m_{y-1-x}$ is true, and v's core number will not exceed $y - 1 + m_{y-1-x}$. We can discover that the core numbers of vertices equalling y are affected by the insertion of S_{y-1}. For a vertex v satisfying $y \le C_G(v) < y + m_{y-x}$, it may change to $\max(y + m_{y-1-x} - 1, y + m_{y-x})$. Since a vertex v with core number k will be affected by the insertion of edge sets $S_k, S_{k-1}, \ldots, S_x$, v's core number will not exceed $\max(k + m_{k-x}, k - 1 + m_{k-1-x}, \ldots, x + m_0) = k + \max_{0 \le i \le k-x}(-k + x + i + m_i) = k + t_{k-x}$. For a vertex v satisfying $x \le C_G(v) < y + t_{y-x}$, its updated core number must be equal to or smaller than $C_G(v) + t_{C_G(v)-x}$. For a vertex v, if $C_G(v) < x$ or $C_G(v) \ge y + t_{y-x}$, from Lemma 3, we know its core number will not change.

Based on Lemma 4, we can directly calculate the initial h-indices based on the core numbers of G before insertion. Since this method needs to access all vertices in the graph, we call it the Global algorithm.

2) Global Algorithm: At the beginning of the algorithm, we set $work\_list$ as $V(S)$. Then, we find the minimum core number (corresponding to x in Lemma 4) and the maximum core number (corresponding to y in Lemma 4) of the inserted edges (lines 4-5). For a vertex $v \in V(S)$, we update its $nise(v)$ (lines 6-8). Since the vertices are evenly distributed on different machines, we need to synchronize and take the minimum (maximum) value to get the correct result (lines 9-10). In line 12, t is an array and $t(j)$ corresponds to t_j in Lemma 4. We update t according to Lemma 4 (lines 11-16). For a vertex v, if its core number has been changed, we add v to the new $work\_list$ (lines 18-21). We compute the final core numbers of all vertices using Algorithm 1.

Performance Analysis. The Global algorithm has an additional round than Algorithm 1. Notably, the complexity of Algorithm 1 is predicated on two distinct parameters, namely, $h^0(v)$ and $work\_list$, both of which are derived from Lemma 4. Consequently, in analyzing the performance of the Global algorithm, it becomes necessary to employ the parameters explicated in Lemma 4.

Theorem 5. Given a graph $G = (V(G), E(G))$, for a vertex $v \in V(G)$, $h^0(v)$ is calculated according to Lemma 4. The round complexity of Algorithm 3 is $O(\sum_{v \in V(G), x \le C_G(v) < y + t_{y-x}} C_G(v) + t_{C_G(v)-x} - C_{G_I(S)}(v))$.

Proof. In the worst case, just one vertex's h-index changes per round, and the changing size is one. The needed number of rounds of vertex v is $h^0(v) - C_G(v) = C_G(v) + t_{C_G(v)-x} - C_{G_I(S)}(v)$ while $x \le C_G(v) < y + t_{y-x}$. In other case, $h(v)$ will not change. Since only one vertex's core number changes in each round, the total number of rounds needed for Algorithm 1 is the sum of the number of rounds required for each vertex. So the total round is bounded by $\sum_{v \in V(G), x \le C_G(v) < y + t_{y-x}} C_G(v) + t_{C_G(v)-x} - C_{G_I(S)}(v)$.

Algorithm 3: Global($G_I(S), h, work\_list$).

```
   Input  : G_I(S), core numbers of G, V(S)
   Output: core numbers of G_I(S)
1  x ← infinity;
2  y ← 0;
3  for v ∈ work_list do
4      x ← min(x, h(v));
5      y ← max(y, h(v));
6      for u ∈ neb(v) do
7          if u ∈ work_list and h(u) ≥ h(v) then
8              nise(v) ← nise(v) + 1;
9  sync x, take the minimum value;
10 sync y, take the maximum value;
11 for v ∈ work_list do
12     t(h(v)) ← max(t(h(v)), nise(v) + h(v));
13 sync t, take the maximum value;
14 for i ← 1 up to y − x do
15     if t(i − 1) > 0 then
16         t(i) ← max(t(i), t(i − 1) − 1);
17 work_list ← empty, all_list ← V(G);
18 for v ∈ all_list do
19     if h(v) ≥ x and h(v) < y + t(y − x) then
20         h(v) ← min(h(v) + t(h(v) − x), deg(v));
21         work_list.set(v)
22 for v ∈ work_list do
23     sync h(v)
24 return Async-H-Index(G_I(S), h, work_list);
```

Theorem 6. Given a graph $G = (V(G), E(G))$, for a vertex $v \in V(G)$, $h^0(v)$ is calculated according to Lemma 4. Suppose there are n machines in total, the message complexity of Algorithm 3 is $O(\sum_{v \in V(G), x \le C_G(v) < y + t_{y-x}} (n-1) * (C_G(v) + t_{C_G(v)-x} - C_{G_I(S)}(v)))$.

Proof. For a vertex $v \in V(G)$, only when $h(v)$ decreases, v will be added to $send\_list$. In the worst case, v will be added to $send\_list$ with $h^0(v) - C_G(v) = C_G(v) + t_{C_G(v)-x} - C_{G_I(S)}(v)$ times. In Gemini, if v is in $send\_list$ and is located on machine i, then Gemini will send $h(v)$ to all machines except machine i. Because there are n machines in total, the number of messages v sent is $(n-1) * t_{C_G(v)-x}$. So the message complexity of Algorithm 3 is bounded by $\sum_{v \in V(G), x \le C_G(v) < y + t_{y-x}} (n-1) * (C_G(v) + t_{C_G(v)-x} - C_{G_I(S)}(v))$.

In Algorithm 3, we update the initial h-indices according to Lemma 4 that requires to access all vertices in the graph (line 18). The vertices whose core numbers have been changed are connected to the insertion edges. But Algorithm 3 does not take advantage of this observation. Take Fig. 3 as an example, after $e = (x_1, y_1)$ is inserted into the graph, Algorithm 3 will set all vertices' initial h-indices to 2, resulting in multiple iterations.

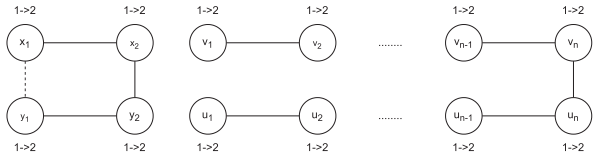


Fig. 3. $e = (x_1, y_1)$ is the edge to be inserted. The left number on top/bottom of each vertex represents its core number before e's insertion. The right number on top/bottom of each vertex represents its initial h-index calculated by the Global algorithm after e's insertion.

If we want to take advantage of the above observation, we need to traverse from the insertion edges to find connected vertices. The traversal reduces the number of rounds for convergence but also has additional overhead to find initial h-indices. We will design an algorithm that takes advantage of the connectivity observation, and our inspiration comes from the TRAVERSAL algorithm [2]. Our algorithm will perform multiple TRAVERSALs in a pipelined fashion.

3) Pipeline Algorithm: The TRAVERSAL algorithm is frequently used to design multiple-edge processing algorithms for core maintenance [3], [5], [6], [9]. Suppose edge $e = (u, v)$ is an inserted edge, $C_G(e) = k$, and $C_G(u) \leq C_G(v)$. The TRAVERSAL algorithm traverses the reachable tree of vertex u. Suppose the current traversed vertex is v. If $sd(v) > C_G(v)$, and $C_G(v) = C_G(u)$, the algorithm will add v to the candidate set, which is a set of vertices whose core numbers may change. If $sd(v) \leq C_G(v)$, which indicates that vertex v cannot be contained in $(k+1)$-core, the algorithm will start eviction from vertex v and evict the vertices that cannot become $(k+1)$-core owing to vertex v. When no vertices need to be evicted, the core numbers of the remaining vertices in the candidate set are increased by one.

Suppose we insert the edges set S into G, and each vertex's core number changes by at most one. In this case, we can run multiple TRAVERSALs concurrently without impacting the final result [3]. However, a vertex may be traversed by more than one TRAVERSAL, making it more difficult to judge whether we can add this vertex to the candidate set. To solve this issue, [3], [5], [6], [9] require a core number corresponds to a TRAVERSAL. Performing in this way undoubtedly reduces parallelism. Since we only care about the initial h-indices, we will not perform the eviction. If we do not evict vertices in the candidate set, we can determine whether a vertex can be added to the candidate set when it is first traversed. When a vertex $v \in V(G)$ is first traversed, only if $sd(v) > C_G(v)$, v will be added to the candidate set.

For a vertex v, we directly increase v's initial h-index by one when v is added to the candidate set. We can find that when two TRAVERSALs starting from the same vertex are separated by one round, the candidate set will also be updated correctly. So we can execute the TRAVERSAL algorithm in a pipelined fashion.

The detailed algorithm is listed in Algorithm 4. In the beginning, $work\_list$ is set to $V(S)$. The algorithm computes $nise(v)$ of each vertex $v \in V(S)$ (lines 1-4) where $nise(v)$ represents the maximum number of TRAVERSALs starting from vertex v.

Algorithm 4: Pipeline($G_I(S)$, h, $work\_list$).

```
   Input  : G_I(S), core numbers of G, V(S)
   Output: core numbers of G_I(S)
1  for v ∈ work_list do
2      for u ∈ neb(v) do
3          if u ∈ work_list and h(u) ≥ h(v) then
4              nise(v) ← nise(v) + 1;
5  preH ← copy of h;
6  while work_list is not empty do
7      send_list ← empty, re_list ← empty;
8      for v ∈ work_list do
9          sd(v) ← computeSd(neb(v), preH);
10         if nise(v) ≠ 0 then
11             if round  mod  2 = 1 then
12                 re_list.set(v);
13             else
14                 if nise(v) > 0 and sd(v) > h(v) then
15                     h(v) ← h(v) + 1;
16                     re_list.set(v);
17                     send_list.set(v);
18                     change_list.set(v);
19                     nise(v) ← nise(v) − 1;
20                 else
21                     nise(v) ← 0;
22         else
23             if sd(v) > h(v) then
24                 h(v) ← h(v) + 1;
25                 send_list.set(v);
26                 change_list.set(v);
27     for u ∈ send_list do
28         sync h(u);
29         for v ∈ neb(u) do
30             if preH(v) = h(u) − 1 and h(v) = preH(v)
               then
31                 re_list.set(v);
32     for u ∈ send_list do
33         preH(u) ← h(u);
34     work_list ← re_list;
35 return Async-H-Index(G_I(S), h, change_list);
```

The $preH$ array is the copy of the h array in the last round (line 5). The $work\_list$ is a set of vertices that need to be calculated in the current round (line 6). The $send\_list$ is a set of vertices whose h-indices change in the current round (line 7). The re_list is a set of vertices that need to be calculated in the next round (line 7). The $change\_list$ is a set of vertices whose h-indices changed (line 18).

We start a Breadth-First-Search (BFS) from the vertices of $V(S)$ to find the candidate set. If a vertex $v \in V(S)$ is still possible to start BFS from this vertex and the number of rounds is even, we increase $h(v)$ by one, and add v to re_list, $send\_list$ and $change\_list$ (lines 14-19). If the number of rounds is odd, we only add v to re_list to separate the TRAVERSALs starting from vertex v (line 11-12). For the vertex v where $nise(v)$

Algorithm 5: computeSd($neb(v), preH$).

Input : The neighbors of vertex v, $preH$
Output: $sd(v)$
1 $sd(v) \leftarrow 0$;
2 for $u \in neb(v)$ do
3 if $preH(u) \geq preH(v)$ then
4 $sd(v) \leftarrow sd(v) + 1$;
5 return $sd(v)$;

Algorithm 6: Decremental($G_D(S)$, h, S).

Input : $G_D(S)$, core numbers of G, S
Output: core numbers of $G_D(S)$
1 $work\_list \leftarrow empty$;
2 for $v \in V(S)$ do
3 $work\_list.set(v)$
4 return Async-H-Index($G_D(S)$, h, $work\_list$);

$= 0$ and $sd(v) > h(v)$, we increase $h(v)$ by one and add v into $send\_list$ and $change\_list$ (lines 23-26).

For vertex $v \in send\_list$, we add its neighbors with the same core number to re_list (lines 26-30). It should be noted that the $h(v)$ has been changed in the current round. We only need to add vertex v's neighbors whose h-indices are the same as $h(v) - 1$ and have not changed in the current round to re_list (lines 29-30). We compute the final core numbers of all vertices using Algorithm 1 (line 35).

Performance Analysis. The Pipeline algorithm necessitates a traversal process to update the initial h-index, thus warranting an additional analysis of the complexity associated with the computation of the initial h-index. It is worth noting, however, that despite this modification, the initial h-index in this algorithm still conforms to the conditions outlined in Lemma 4, thereby implying that the complexity of the convergence stage is akin to that of the Global algorithm.

Theorem 7. Suppose S is the inserted edges set, the round complexity of Algorithm 4 is $O(max_{v\in V(S)}(2 * nise(v) + D(Tree(v))) + \sum_{\bigcup_{v\in V(s)} Tree(v)}(n-1) * (C_G(v) + t_{C_G(v)-x} - C_{G_I(S)}(v)))$.

Proof. Initially, an investigation will be conducted into the number of rounds needed for determining the initial h-index. For a vertex $v \in V(S)$, v will start at most $nise(v)$ times TRAVERSALs. $\max_{v\in V(S)} 2 * nise(v)$ rounds are required to ensure that all vertices in $V(S)$ no longer perform TRAVERSAL. Only vertices whose core numbers equal $h(v)$ can be reached and will change their core numbers. Therefore, the round complexity is bounded by $max_{v\in V(S)} 2 * nise(v) + D(Tree(v))$. $Tree(v)$ represents the union of the reachable trees of TRAVERSALs starting from vertex v, and $D(Tree(v))$ refers to the depth of $Tree(v)$.

The number of rounds analysis in the convergence phase is similar to the Global algorithm, except by replacing $work\_list$ with $\bigcup_{v\in V(s)} Tree(v)$. So the total round is bounded by $max_{v\in V(S)}(2 * nise(v) + D(Tree(v))) + \sum_{\bigcup_{v\in V(s)} Tree(v)}(n-1) * (C_G(v) + t_{C_G(v)-x} - C_{G_I(S)}(v))$.

Theorem 8. Suppose S is the inserted edges set, and there are n machines. The message complexity the Algorithm 4 is $O(\sum_{\bigcup_{v\in V(s)} Tree(v)}(n-1) * (2 * t_{C_G(v)-x} + C_G(v) - C_{G_I(S)}(v)))$.

Proof. Primarily, an assessment will be undertaken regarding the number of messages necessary for computing the initial h-index. According to Lemma 4, for a vertex $v \in V(G)$, the change of vertex v's core number cannot exceed $C_G(v) - t_{C_G(v)-x} - C_G(v) = t_{C_G(v)-x}$. In the worst case, each vertex v will change $t_{C_G(v)-x}$ times. For each change, the message of v will be sent to $n-1$ machines, so the message complexity is bounded by $\sum_{\bigcup_{v\in V(s)} Tree(v)}(n-1) * t_{C_G(v)-x}$.

The number of messages analysis in the convergence phase is similar to the Global algorithm, except by replacing $work\_list$ with $\bigcup_{v\in V(s)} Tree(v)$. So the total number of messages is bounded by $\sum_{\bigcup_{v\in V(s)} Tree(v)}(n-1) * (2 * t_{C_G(v)-x} + C_G(v) - C_{G_I(S)}(v))$.

B. Decremental Core Maintenance

Similarly, processing deletion needs two steps: (1) compute the initial h-indices; (2) convergence. However, since the core numbers before deletion have already been treated as initial h-indices, it is unnecessary to compute the initial h-indices. Now we face the challenge of identifying which vertices need to perform convergence. The Global algorithm and the Pipeline algorithm add the vertices whose core numbers may change to the $work\_list$. But for deletion, we cannot rapidly determine the vertices whose core numbers may change. In fact, we need only to add the vertices in $V(S)$ to the $work\_list$.

Lemma 9. Given a graph $G = (V(G), E(G))$, if the edges set S is deleted from G, we add vertex $v \in V(S)$ into $work\_list$, and $h(v)$ is set as $C_G(v)$. After we implement Algorithm 1, we have $h(v) = C_{G_D(S)}(v)$, where v is a vertex in $G_D(S)$.

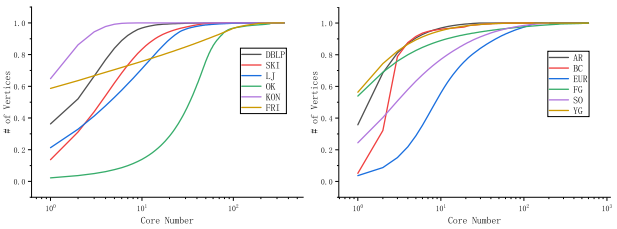
Proof. According to Algorithm 1, if all vertices in $G_D(S)$ are added to the $work\_list$, we conclude $h(v) = C_{G_D(S)}(v)$ is true after implementing Algorithm 1. We conduct one convergence for each vertex in $V(G)$ and assume the set of vertices whose h-indices change is V_1. Similarly, if the $work\_list$ is $V(S)$, we do one convergence for vertex $v \in V(S)$, and add the vertices whose h-indices change into the vertices set V_2. Next, we will prove that $V_1 = V_2$.

Since $V(S) \subseteq V(G)$, we need to examine two cases: (I) suppose a vertex $v \in V(G)$ and $v \in V(S)$. If v is added to V_1, due to the same calculation, v must also be added to V_2; (II) suppose a vertex $v \in V(G)$ and $v \notin V(S)$. If u is a neighbor of v and $h(u) = C_G(u)$, we know no neighbors of v ($v \notin V(S)$) are added or eliminated, so $h(v)$ will not change. Therefore, $V_1 = V_2$ is true. So whether the $work\_list$ is $V(S)$ or $V(G)$, Algorithm 1 already has the same state in the second round.

Performance Analysis. The Decremental algorithm resembles the Global algorithm, except for the variation in the value inputted into Algorithm 1. Thus, to determine the analysis results of the Decremental algorithm, it suffices to substitute the parameters of the analysis of the Global algorithm.

TABLE III
DATASETS

Dataset	$\|V\|$	$\|E\|$	avg.deg
DBLP	0.3M	1.0M	7
AR(amazon-ratings)	2.2M	5.7M	5
YG(youtube-growth)	3.3M	9.4M	6
SKI(skitter)	1.7M	11.1M	13
EUR(epinions-user-ratings)	0.76M	13.7M	36
FG(flickr-growth)	2.3M	22.9M	20
SO(stack-overflow)	2.6M	28.2M	22
LJ(liveJournal)	4.0M	34.7M	17
BC(bitcoin)	24.6M	86.1M	7
KON(konect)	59.2M	92.5M	3
OK(orkut)	3.1M	117.2M	76
FRI(frinedster)	65.6M	1806.1M	55



(a) static datasets (b) temporal datasets

Fig. 4. Core numbers distribution for datasets.

Theorem 10. Suppose the edges set S is deleted from the graph G, the round complexity of Algorithm 6 is $O(\sum_{v\in V(G_D(S))}(C_G(v) - C_{G_D(S)}(v)))$.

Theorem 11. Suppose the edges set S is deleted from the graph G, the message complexity of Algorithm 6 is $O(\sum_{v\in V(G_D(S))}((n-1) * (C_G(v) - C_{G_D(S)}(v))))$.

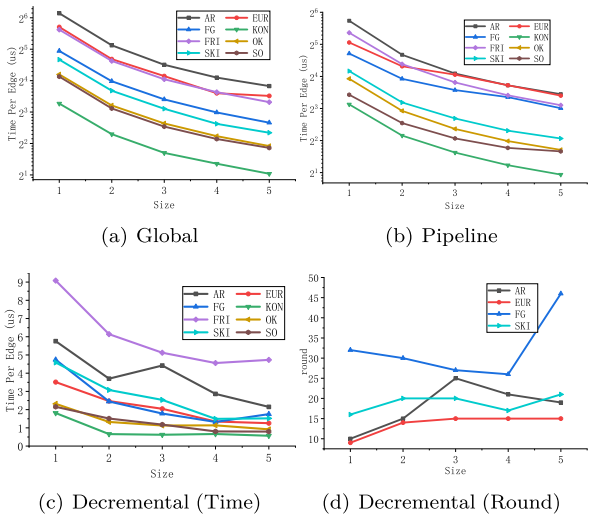
VI. EXPERIMENT

Our experiments were performed on a 5-machine high-performance cluster connected through a private network (100 Gbps bandwidth). Each machine hosts two Intel Xeon Gold 5117 CPUs (each with 28 cores) and 256 GB of DRAM. As mentioned, our algorithms are implemented with Gemini. The source code of the experiment can be found here[2].

The algorithms read two files from the disk: the original graph and the inserted/deleted edges. We remove these edges from the original graph to test the Decremental algorithm and then reinsert them to test the Global and Pipeline algorithms. In addition, we disregard the loading time of the file and only calculate the execution time of the algorithms.

Datasets: Table III displays twelve datasets from the real world that may be viewed and downloaded through NDR[3] and SNAP[4]. DBLP, SKI, LJ, KON, OK, and FRI are static, whereas others are temporal. To satisfy our algorithm's requirements, we convert the directed graphs contained in the datasets to undirected graphs, with the corresponding transformation of attribute within Table III. Fig. 4 illustrates the distribution of core numbers in the above datasets. For most datasets, the core numbers of more than ninety percent of the vertices are less than 100.

[2]https://qiangshenghua.github.io/papers/distributedcore.zip
[3]https://networkrepository.com/index.php
[4]http://snap.stanford.edu/data/



(a) Global (b) Pipeline
(c) Decremental (Time) (d) Decremental (Round)

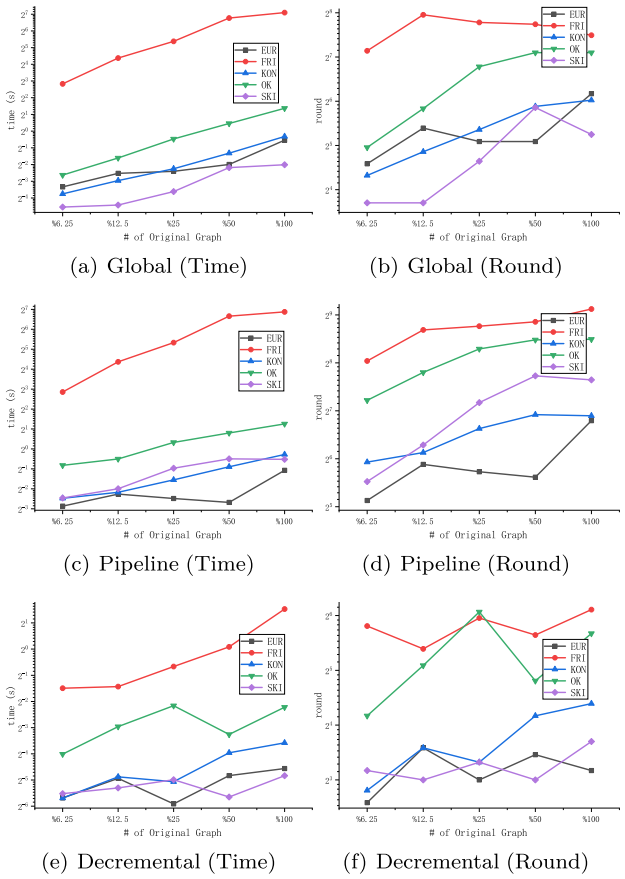
Fig. 5. Stability of algorithms (the average time spent on each edge across various datasets).

A. Stability Evaluation

Practical applications often involve the core maintenance of numerous graphs. Hence, a crucial inquiry is whether an algorithm's performance remains stable when applied to distinct datasets. In this subsection, we evaluated the stability of our algorithms using eight datasets, namely AR, EUR, FG, KON, FRI, OK, SKI, and SO. For the edge set of insertion and deletion, we opted for i‰of the corresponding dataset, where $1 \leq i \leq 5$. Notably, for temporal graphs, we chose the most recently updated edges, whereas for static graphs, we randomly selected the set of edges. The experiment was conducted on four machines, with 48 threads per machine. The experimental outcomes are depicted in Fig. 5, with the x-axis representing the size of the inserted/deleted edge set and the y-axis indicating the average cost time on each edge.

Fig. 5(a), (b), and (c) illustrate that the ratio of the maximum average time spent per edge to the minimum average time at a given x-coordinate is less than 16. Furthermore, as the magnitude of the inserted/deleted edge set escalates, the average time per edge diminishes, signifying that our algorithm performs better under larger edge sets. This is attributed to the superior parallel processing capabilities of the h-index upon which our method is based.

Concurrently, we discovered that in the evaluation of the Decremental algorithm, the average time expended on each edge for some datasets (AR, FG) does not exhibit a complete reduction with the expansion of the deleted edge set. Consequently, we investigated the number of rounds involved in the deletion case for four datasets, AR, EUR, FG, and SKI, as illustrated in Fig. 5(d). Our findings revealed that the number of rounds had undergone significant alterations for the AR and FG datasets. As the calculation of vertices within each round of the Decremental algorithm is small, the impact of the number of rounds on performance is more pronounced. Hence, the Decremental algorithm's cost time does not decline entirely with the increase in the deleted edge set.

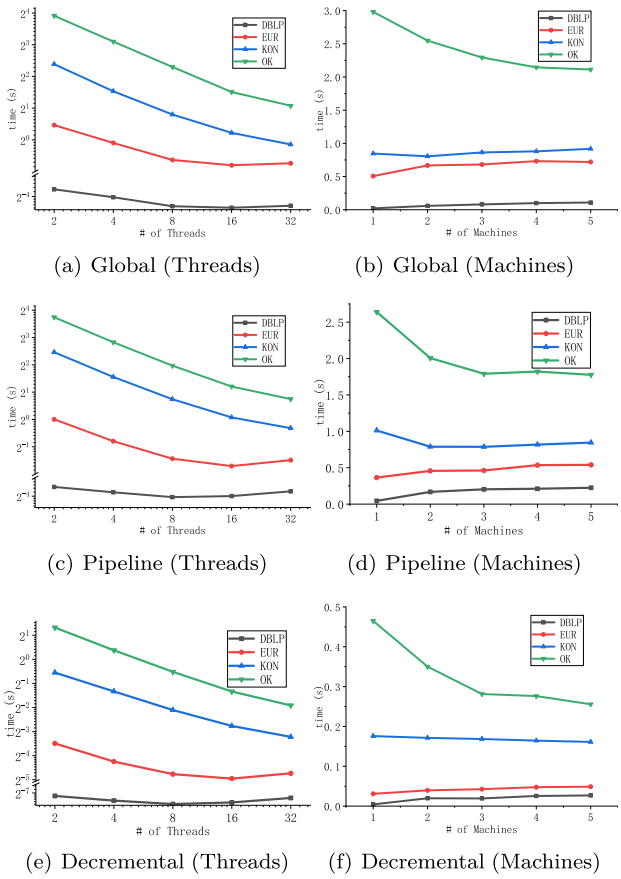


(a) Global (Time) (b) Global (Round)

(c) Pipeline (Time) (d) Pipeline (Round)

(e) Decremental (Time) (f) Decremental (Round)

Fig. 6. Scalability of algorithms (time or rounds required for different graph sizes).



(a) Global (Threads) (b) Global (Machines)

(c) Pipeline (Threads) (d) Pipeline (Machines)

(e) Decremental (Threads) (f) Decremental (Machines)

Fig. 7. Parallelism of algorithms (execution time under varying numbers of threads or machines).

B. Scalability Evaluation

In this subsection, we evaluated the algorithms' scalability by adjusting the graph size. Our assessment focused on five datasets: EUR, FRI, KON, OK, and SKI. Specifically, we created new datasets based on the original datasets, with the size of 6.25%, 12.5%, 25%, 50%, and 100% of original datasets, and selected 1‰edges of the corresponding graph for insertion/deletion. The experiment was conducted on four machines, with 48 threads per machine. The experimental findings are presented in Fig. 6, where the x-axis denotes the size of the produced graph, and the y-axis indicates the time or number of rounds cost by the algorithms.

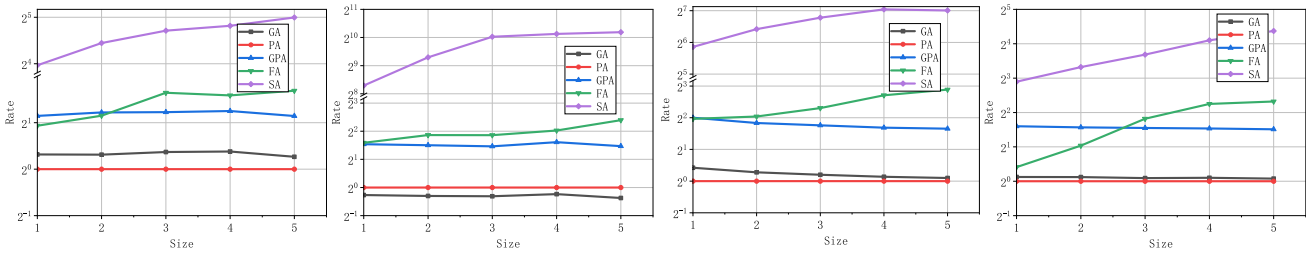
According to the data depicted in Fig. 6(a) and (c), it can be observed that with an increase in the size of the graph, both the Global and Pipeline algorithms demonstrate a proportional increase in their respective cost time. This outcome aligns with our expectations, given that processing large graphs requires more time. Moreover, we investigated the number of rounds cost by Global and Pipeline algorithms, shown in Fig. 6(b) and (d), which indicated that the number of rounds exhibits no substantial correlation with the graph size. As demonstrated in Fig. 3, the number of rounds primarily depends on the graph's structure.

In distributed algorithms, the number of rounds and the cost time are typically positively related, but the experimental findings do not always conform to this norm. This is because in the Global algorithm and the Pipeline algorithm, a substantial number of vertices must be calculated during the initial few rounds of convergence, necessitating more time. As such, the time spent during each round is non-uniform.

As depicted in Fig. 6(e) and (f), the cost time of executing the Decremental algorithm is positively correlated with the number of rounds required. This can be attributed to the fact that during each round of the Decremental algorithm, fewer vertices are computed, resulting in a more consistent time spent during each round. Consequently, for the Decremental algorithm, the cost time is primarily contingent on the number of rounds rather than the graph size.

C. Parallelism Evaluation

In this subsection, we conducted experiments to investigate the impact of the number of threads and machines on the performance of the algorithms. We utilized four data sets: DBLP, EUR, KON, and OK. For each of these data sets, we selected 1‰of edges as the edge set to be inserted/deleted. Specifically, we tested the algorithms on four machines and adjusted the number of threads per machine to 2, 4, 8, 16, and 32. Additionally, we fixed the number of threads to 48 and varied the number of machines from 1 to 5. The experimental results are presented in Fig. 7, where the x-axis represents the number of threads or



(a) AR (b) BC (c) FG (d) OK

Fig. 8. Comparison with the shared memory algorithms (Insertion).

machines, and the y-axis represents the time required to execute the algorithms.

Based on the results shown in Fig. 7(a), (c) and (e), it can be observed that the execution time of the algorithms generally decreases as the number of threads increases. However, for the relatively small graphs DBLP and EUR, using 32 threads results in a longer execution time than 16 threads do for the algorithms. This is because increasing the number of threads can not only reduce computational overhead, but also increase management overhead, so using more threads is not always the optimal solution.

From Fig. 7(b), (d) and 7(f), it can be observed that the time taken to execute the algorithm displays two trends as the number of machines increases. In the case of larger graphs, OK and KON, the time required initially decreases and then stabilizes slightly rises. However, for smaller graphs such as DBLP and EUR, the execution time continuously increases as the number of machines increases. This can be attributed to the fact that the increase in the number of machines not only increases the processing power, but also leads to higher communication overhead. The execution time of the algorithm is the sum of calculation and communication time. More processors are generally required for large graphs to reduce the calculation time, hence leading to the initial decrease in execution time. On the other hand, adding more machines increases the communication overhead for smaller graphs, resulting in a slight increase in the overall time.

D. Comparisons With Existing Algorithms

This subsection presents a performance evaluation of three algorithms: the Global algorithm (GA), the Pipeline algorithm (PA), and the Decremental algorithm (DA). Furthermore, they are compared with shared memory parallel algorithms and private memory distributed algorithms.

The following evaluation compares our algorithms against shared memory parallel algorithms, including shared memory parallel hypergraph core maintenance algorithms using h-index (GPA) [28], parallel algorithms based on joint edge sets (FA) [6], and parallel algorithms based on superior edge sets (SA) [4]. The pregel-like algorithm GPA is implemented with Gemini to ensure a fair evaluation. For FA and SA algorithms, their approach is unsuitable for pregel-like, and thus we directly use their source codes. This evaluation was conducted on a single machine with 48 threads.

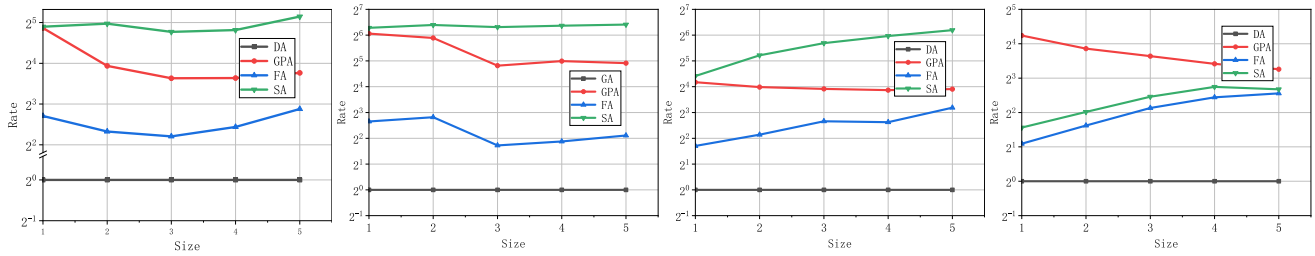
In this evaluation, we utilized four data sets: AR, BC, FG, and OK. For the temporal graphs, the most recent i‰edges are selected for insertion/deletion, where $1 \leq i \leq 5$. For static graphs, we randomly select i‰edges. The results of our experiments are presented in Figs. 8 and 9. The x-axis represents the size of the inserted/deleted edge set, while the y-axis denotes the cost time ratio of various algorithms over either the Pipeline algorithm or the Dcremental algorithm.

Based on the results shown in Figs. 8 and 9, our algorithms outperform the other algorithms in all cases. For the insertion case, the maximum speedup ratio of our algorithm is 4, compared to GPA. Similarly, compared to FA and SA, the maximum speedup ratio of our algorithm is 8 and 1000, respectively. The experimental results also demonstrate that as the set of inserted edges increases, the speedup ratio of our algorithm compared to FA and SA also increases. This observation suggests that our algorithms are more suitable for handling multi-edge insertion/deletion due to their basis on the h-index, which exhibits better parallelism. Additionally, our algorithm's speedup ratio is relatively stable compared to the GPA algorithm, which is also based on the h-index.

In most cases examined, we observed that the Pipeline algorithm outperforms the Global algorithm, except for BC. It is noteworthy that both the Pipeline algorithm and the SA algorithm are implemented based on the TRAVERSAL algorithm. Our examination revealed that the SA algorithm consumes considerable time on BC, indicating that the graph traversal process takes a significant amount of time, thus contributing to the increased time required by the Pipeline algorithm.

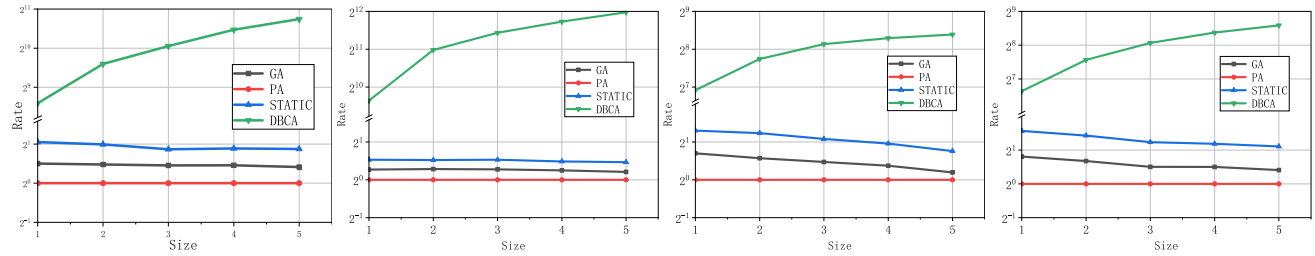
Additionally, the performance comparison is extended to private memory distributed algorithms, which includes the distributed core decomposition algorithm (STATIC) [10], and the state-of-the-art distributed core maintenance algorithm (DBCA) [9]. These two pregel-like algorithms are implemented with Gemini to ensure a fair evaluation. This evaluation was conducted on four machines, with 48 threads per machine. In this evaluation, we utilized four data sets: LJ, OK, SO, and YG.

Based on the results shown in Figs. 10 and 11, our proposed algorithms consistently demonstrate the lowest execution time. Specifically, compared to the DBCA algorithm, our algorithms achieve a speedup ratio of over 100 in cases involving insertion and deletion. Additionally, as the size of the edge set for



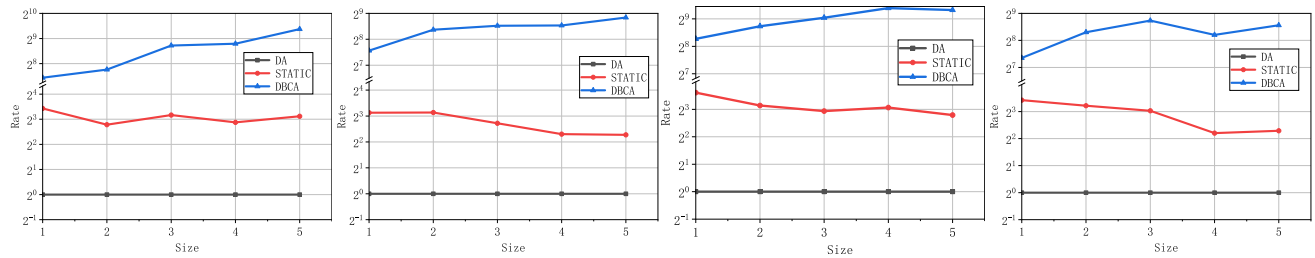
(a) AR (b) BC (c) FG (d) OK

Fig. 9. Comparison with the shared memory algorithms (Deletion).



(a) LJ (b) OK (c) SO (d) YG

Fig. 10. Comparison with the distributed algorithms (Insertion).



(a) LJ (b) OK (c) SO (d) YG

Fig. 11. Comparison with the distributed algorithms (Deletion).

insertion and deletion increases, the speedup ratio of DBCA also increases, indicating that our algorithm is better suited for handling multi-edge insertion and deletion. It is worth noting that the average speedup ratio achieved by our algorithm compared to DBCA is greater than the average speedup ratio achieved compared to FA and SA. The reduction in parallelism by DBCA can be attributed to its need to conform to the distributed environment. Although its underlying principles share similarities with FA and SA, DBCA has established more stringent criteria for the edge set of insertion and deletion in order to accommodate the distributed setting.

Our algorithms achieve the average speedup ratio of 2 and 8 for insertion and deletion, compared to the STATIC algorithm. As the size of the insertion/deletion edge set increases, the speedup ratio decreases, as expected. This is because the maintenance algorithm requires more time to process larger edge sets, bringing it closer to the performance of the static decomposition algorithm.

VII. Related Work

There are two different versions of distributed core decomposition algorithms: (I) the peeling algorithms [8], [17]: these algorithms first remove the vertices with the lowest degrees, then repeat the process until all vertices have been removed. The core number of v is equal to the degree of v when it is removed; (II) the h-index [1] based algorithm [10]: each vertex updates its core number based on its neighbors' core numbers. When there are no vertices whose core numbers change, the algorithm terminates. Compared with the peeling algorithm, the h-index based algorithm has a higher degree of parallelism and is more suitable for distributed environments.

Since most graph applications are dynamic, core maintenance attracts increasing attention. Sariyuce et al. [2] found that the vertices' core numbers may change by one when a single edge is inserted/deleted. Based on this, a centralized single-edge core maintenance algorithm was proposed. Zhang et al. [3] presented

an order-based technique for accelerating the centralized single-edge core maintenance algorithm.

Wang et al. [4], Jin et al. [5], and Hua et al. [6] devised parallel multi-edge algorithms based on centralized single-edge algorithms [2]. They have various requirements for the inserted/deleted edges sets. If the inserted edges sets do not fulfill the requirements, then the edges sets will be partitioned into multiple eligible edges sets which will be performed sequentially. Gabert et al. [28] proposed an h-index-based shared memory and parallel core maintenance algorithm on hypergraphs, which differs from our algorithm in computing the initial h-index.

Considering that centralized algorithms may have the memory issues, Wen et al. [7] proposed centralized I/O efficient single-edge algorithms. Although this method can somewhat alleviate the memory issue, distributed algorithms might be alternative solutions when the graph size is too large. Using a technique similar to [2], Aridhi et al. [8] constructed distributed single-edge processing algorithms based on various distributed systems. When there are multiple inserted/deleted edges, the above algorithms become inefficient. Weng et al. [9] presented a distributed algorithm that can simultaneously handle multiple edges with different core numbers.

VIII. Conclusion

This article aims to tackle the core maintenance problem in a distributed manner. Existing distributed core maintenance algorithms [8], [9] are all based on traversal and cannot simultaneously process the edges with the same core number, which leads to a quick increase in the cost time when multiple edges are inserted/deleted.

Inspired by the h-index [1] based distributed core decomposition algorithm [10] which can circumvent the above issues and have a high degree of parallelism, we propose two h-index based distributed core maintenance algorithms: the Global algorithm and the Pipeline algorithm. Note that directly applying the distributed core decomposition algorithm in dynamic graphs will entail all vertices' recalculations upon graph changes. Our distributed core maintenance algorithms can avoid this issue by presenting two techniques to identify the vertices whose core numbers may change and to reduce the initial h-indices in the convergence stage. The Global algorithm calculates the initial h-indices without considering the graph connectivity and the Pipeline algorithm takes advantage of it. The Pipeline algorithm outperforms the Global Algorithm in most cases, but underperforms the latter in graphs with a time-consuming traversal processing. Compared to the state-of-the-art distributed maintenance algorithm [9], in the case of both insertion and deletion, the time speedup ratio is more than 100.

There are also some potential points in our work which warrant further research. First, the algorithm introduced in this paper is inspired by the TRAVERSAL algorithm [2] in cases of single edge insertion/deletion. We notice that there are also some other algorithms with excellent performance which have not been employed in a distributed manner, such as the order-based algorithm proposed in [3]. Exploring the parallelization of these presents an intriguing avenue for future research. Another possible future work is to find more efficient methods to lower the initial h-index in the convergence step to further reduce the cost time. The hierarchical core definition [27] which reflects the connectivity of the k-core subgraph might be a potential direction. Note that maintaining the hierarchical core numbers will also bring additional overheads. Finally, efficient distributed core maintenance algorithms for dynamic hypergraphs [28], [29], [30], [32] have been receiving an increasing attention. Extending our methods to hypergraphs is a tempting direction.

References

[1] L. Lü et al., "The h-index of a network node and its relation to degree and coreness," *Nature Commun.*, vol. 7, no. 1, pp. 1–7, 2016.

[2] A. E. Sarıyüce et al., "Incremental K-core decomposition: Algorithms and evaluation," *VLDB J.*, vol. 25, no. 6, pp. 425–447, 2016.

[3] Y. Zhang, J. X. Yu, Y. Zhang, and L. Qin, "A fast order-based approach for core maintenance," in *Proc. IEEE 33rd Int. Conf. Data Eng.*, 2017, pp. 337–348.

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

[5] H. Jin, N. Wang, D. Yu, Q. -S. Hua, X. Shi, and X. Xie, "Core maintenance in dynamic graphs: A parallel approach based on matching," *IEEE Trans. Parallel Distrib. Syst*, vol. 29, no. 11, pp. 2416–2428, Nov. 2018.

[6] Q.-S. Hua et al., "Faster parallel core maintenance algorithms in dynamic graphs," *IEEE Trans. Parallel Distrib. Syst*, vol. 31, no. 6, pp. 1287–1300, Jun. 2020.

[7] 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.

[8] 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.

[9] T. Weng, X. Zhou, K. Li, P. Peng, and K. Li, "Efficient distributed approaches to core maintenance on large dynamic graphs," *IEEE Trans. Parallel Distrib. Syst*, vol. 33, no. 1, pp. 129–143, Jan. 2022.

[10] 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.

[11] G. Malewicz et al., "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146.

[12] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 301–316.

[13] R. Chen, J. Shi, Y. Chen, B. Zang, H. Guan, and H. Chen, "PowerLyra: Differentiated graph computation and partitioning on skewed graphs," *ACM Trans. Parallel Comput.*, vol. 5, pp. 1–39, 2019.

[14] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph processing in a distributed dataflow framework," in *Proc. 11th USENIX Symp. Operating Syst. Des. Implementation*, 2014, pp. 599–613.

[15] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proc. 10th USENIX Symp. Operating Syst. Des. Implementation*, 2012, pp. 17–30.

[16] R. Dathathri et al., "Gluon-async: A. bulk-asynchronous system for distributed and heterogeneous graph analytics," in *Proc. 28th Int. Conf. Parallel Architectures Compilation Techn.*, 2019, pp. 15–28.

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

[18] H. Qin, R.-H. Li, G. Wang, X. Huang, Y. Yuan, and J. X. Yu, "Mining stable communities in temporal networks by density-based clustering," *IEEE Trans. Big Data*, vol. 8, no. 3, pp. 671–684, Jun. 2022.

[19] L. Lin, P. Yuan, R.-H. Li, and H. Jin, "Mining diversified Top-r lasting cohesive subgraphs on temporal networks," *IEEE Trans. Big Data*, vol. 8, no. 6, pp. 1537–1549, Dec. 2022.

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

[21] C.-Y. Huang, Y.-H. Fu, and C.-T. Sun, "Identify influential social network spreaders," in *Proc. IEEE Int. Conf. Data Mining Workshop*, 2014, pp. 562–568.

[22] J. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani, "Large scale networks fingerprinting and visualization using the k-core decomposition," in *Proc. Adv. Neural Inf. Process. Syst.*, 2005, pp. 41–50.
[23] G. D. Bader and C. W. Hogue, "An automated method for finding molecular complexes in large protein interaction networks," *BMC Bioinf.*, vol. 4, no. 1, pp. 1–27, 2003.
[24] P. Meyer, H. P. Siy, and S. Bhowmick, "Identifying important classes of large software systems through K-core decomposition," *Adv. Complex Syst.*, vol. 17, no. 07–08, 2014, Art. no. 1550004.
[25] H. Zhang et al., "Using the K-core decomposition to analyze the static structure of large-scale software systems," *J. Supercomput.*, vol. 53, pp. 352–369, 2010.
[26] X. Xiong, G. Zhou, Y. Huang, H. Chen, and K. Xu, "Dynamic evolution of collective emotions in social networks: A case study of sina weibo," *Sci. China Inf. Sci.*, vol. 56, no. 7, pp. 1–18, 2013.
[27] Z. Lin, F. Zhang, X. Lin, W. Zhang, and Z. Tian, "Hierarchical core maintenance on large dynamic graphs," in *Proc. VLDB Endowment*, vol. 14, no. 5, pp. 757–770, 2021.
[28] K. Gabert, A. Pinar, and Ü. V. Çatalyürek, "Shared-memory scalable k-core maintenance on dynamic graphs and hypergraphs," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2021, pp. 998–1007.
[29] B. Sun, T.-H. H. Chan, and M. Sozio, "Fully dynamic approximate k-core decomposition in hypergraphs," *ACM Trans. Knowl. Discov. Data*, vol. 14, no. 4, pp. 1–21, 2020.
[30] Q. Luo, D. Yu, Z. Cai, X. Lin, and X. Cheng, "Hypercore maintenance in dynamic hypergraphs," in *Proc. IEEE 37th Int. Conf. Data Eng.*, 2021, pp. 2051–2056.
[31] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.
[32] Q.-S. Hua, X. Zhang, H. Jin, and H. Huang, "Revisiting core maintenance for dynamic hypergraphs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 3, pp. 981–994, Mar. 2023.

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

Hongen Wang received the BE degree from the Huazhong University of Science and Technology, in 2020. He is currently working towards the master's degree with the School of Computer Science and Technology, Huazhong University of Science and Technology. His research interests include dynamic graph algorithms and distributed computing.

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

Xuanhua Shi (Senior Member, IEEE) received the PhD degree in computer engineering from the Huazhong University of Science and Technology, Wuhan, China, in 2005. He is currently a professor with the National Engineering Research Center for Big Data Technology and System Services Computing Technology and System/Services Computing Technology and System Lab, Huazhong University of Science and Technology (China). From 2006, he worked as an INRIA postdoctoral with PARIS team at Rennes for one year. His research interests cloud computing and Big Data processing. He published over more than 100 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, Alibaba, ByteDance, Intel and so on. He is a senior member of CCF.