Distributed D-core Decomposition over Large Directed Graphs

Xuankun Liao*
Hong Kong Baptist University
xkliao@comp.hkbu.edu.hk

Qing Liu*
Hong Kong Baptist University
qingliu@comp.hkbu.edu.hk

Jiaxin Jiang
Hong Kong Baptist University
jxjian@comp.hkbu.edu.hk

Xin Huang
Hong Kong Baptist University
xinhuang@comp.hkbu.edu.hk

Jianliang Xu
Hong Kong Baptist University
xujl@comp.hkbu.edu.hk

Byron Choi
Hong Kong Baptist University
bchoi@comp.hkbu.edu.hk

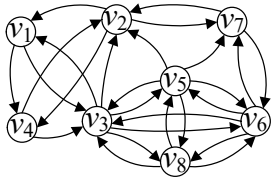
ABSTRACT

Given a directed graph G and integers k and l, a D-core is the maximal subgraph $H \subseteq G$ such that for every vertex of H, its in-degree and out-degree are no smaller than k and l, respectively. For a directed graph G, the problem of D-core decomposition aims to compute the non-empty D-cores for all possible values of k and l. In the literature, several *peeling-based* algorithms have been proposed to handle D-core decomposition. However, the peeling-based algorithms that work in a sequential fashion and require global graph information during processing are mainly designed for *centralized* settings, which cannot handle large-scale graphs efficiently in distributed settings. Motivated by this, we study the *distributed* D-core decomposition problem in this paper. We start by defining a concept called *anchored coreness*, based on which we propose a new H-index-based algorithm for distributed D-core decomposition. Furthermore, we devise a novel concept, namely *skyline coreness*, and show that the D-core decomposition problem is equivalent to the computation of skyline corenesses for all vertices. We design an efficient D-index to compute the skyline corenesses distributedly. We implement the proposed algorithms under both vertex-centric and block-centric distributed graph processing frameworks. Moreover, we theoretically analyze the algorithm and message complexities. Extensive experiments on large real-world graphs with billions of edges demonstrate the efficiency of the proposed algorithms in terms of both the running time and communication overhead.

PVLDB Reference Format:
Xuankun Liao, Qing Liu, Jiaxin Jiang, Xin Huang, Jianliang Xu, and Byron Choi. Distributed D-core Decomposition over Large Directed Graphs. PVLDB, 15(8): 1546 - 1558, 2022.
doi:10.14778/3529337.3529340

1 INTRODUCTION

Graph is a widely used data structure to depict entities and their relationships. In a directed graph, edges have directions to represent links from one vertex to another, which has many real-life applications, e.g., the *following* relationship in online social networks such as Twitter, the *money flow* in financial networks, the *traffic route* in road networks, and the *message forwarding path* in communication networks [26]. Among many graph analysis algorithms, cohesive subgraph analysis is to discover densely connected subgraphs under a cohesive subgraph model. A well-known model used for undirected graphs is k-core, which requires every vertex in the subgraph to have at least k neighbors [39]. As a directed version of k-core, D-core, a.k.a. (k, l)-core, is the maximal directed subgraph such that every vertex has at least k in-neighbors and l out-neighbors within this subgraph [17]. For example, in Figure 1(a), the whole directed graph G is a (2, 2)-core since every vertex has an in-degree of at least 2 and an out-degree of at least 2.



(a) A directed graph G

l \ k	0	1	2	3
0	$v_1v_2v_3v_4$ $v_5v_6v_7v_8$	$v_1v_2v_3v_4$ $v_5v_6v_7v_8$	$v_1v_2v_3v_4$ $v_5v_6v_7v_8$	$v_2v_3v_5v_6$ v_7v_8
1	$v_1v_2v_3v_4$ $v_5v_6v_7v_8$	$v_1v_2v_3v_4$ $v_5v_6v_7v_8$	$v_1v_2v_3v_4$ $v_5v_6v_7v_8$	$v_2v_3v_5v_6$ v_7v_8
2	$v_1v_2v_3v_4$ $v_5v_6v_7v_8$	$v_1v_2v_3v_4$ $v_5v_6v_7v_8$	$v_1v_2v_3v_4$ $v_5v_6v_7v_8$	$v_3v_5v_6v_8$
3	$v_3v_5v_6v_8$	$v_3v_5v_6v_8$	$v_3v_5v_6v_8$	$v_3v_5v_6v_8$

(b) All non-empty D-cores

Figure 1: An example of D-core decomposition on G

As a foundation of D-core discovery, the problem of D-core decomposition aims to compute the non-empty D-cores of a directed graph for all possible values of k and l. D-core decomposition has a number of applications. It has been used to *build coreness-based indexes for speeding up community search* [7, 13], to *measure influence in social networks* [16], to *evaluate graph collaboration features of communities* [17], to *visualize and characterize complex networks* [33], and to *discover hubs and authorities* of directed networks [40]. For example, based on the D-core decomposition results, we can index a graph's vertices by their corenesses using a table [13] or D-Forest [7]; then, D-core-based community search can be accelerated by looking up the table or D-Forest directly, instead of performing the search from scratch [19, 20]. In the literature, *peeling-based* algorithms have been proposed for D-core decomposition in centralized settings [13, 17]. They work in a sequential fashion to remove disqualified vertices one by one from a graph. That is, they first determine all possible values of k (i.e., from 0 to the maximum in-degree of the graph). Next, for each value k, they compute (k, l)-cores for all possible values of l by iteratively deleting the vertices with the smallest out-degree. Figure 1(b) shows the D-core decomposition results of G for different k and l values.

In this paper, we study the problem of D-core decomposition in distributed settings, where the input graph G is stored on a collection of machines and each machine holds only a partial subgraph of G. The motivation is two-folded. First, due to the large size of graph data, D-core decomposition necessitates huge memory space,

*These authors have contributed equally to this work.

Proceedings of the VLDB Endowment, Vol. 15, No. 8 ISSN 2150-8097.
doi:10.14778/3529337.3529340

which may exceed the capacity of a single machine. For example, the existing algorithms could not work for billion-scale graphs due to excessive memory space costs [13]. Second, in practical applications, many large graphs are inherently distributed over a collection of machines, making distributed processing a natural solution [3, 24, 31, 33].

However, the existing peeling-based algorithms are not efficient when extended to distributed settings. In particular, when computing the (k, l)-cores for a given k, the algorithms need to iteratively find the vertices with the smallest out-degree to delete and then update the out-degrees for the remaining vertices, until the graph becomes empty. This process (i) is not parallelizable since the update of out-degrees in each iteration depends on the vertices deletion in the previous iteration and (ii) entails expensive network communications since it needs global graph information.

To address these issues, we design new distributed D-core decomposition algorithms by exploiting the relationships between a vertex and its neighbors. First, inspired by the notion of *k-list* [13], we propose an *anchored coreness*-based algorithm. Specifically, for a vertex v, if we fix the value of k_v, we can compute the maximum value of l_v such that v is contained in the (k_v, l_v)-core. We call this pair (k_v, l_v) an anchored coreness of v. For example, for vertex v_2 in Figure 1, when $k_{v_2} = 0$, the maximum value of l_{v_2} is 2 since $v_2 \in (0, 2)$-core but $v_2 \notin (0, 3)$-core. Hence, $(0, 2)$ is an anchored coreness of v_2. The other anchored corenesses of v_2 are $(1, 2)$, $(2, 2)$, and $(3, 1)$. Once we have computed the anchored corenesses for every vertex, we can easily derive the D-cores from these anchored corenesses. Specifically, given integers k and l, the (k, l)-core consists of the vertices whose anchored coreness (k_v, l_v) satisfies $k_v = k$ and $l_v \geq l$. Thus, the problem of distributed D-core decomposition is equivalent to computing the anchored corenesses in a distributed way. To do so, we first exploit the in-degree relationship between a vertex and its in-neighbors and define an *in-H-index*, based on which we compute the maximum value of k for each vertex. Then, we study the property of $(k, 0)$-core and define an *out-H-index*. On the basis of that, for each possible value of k with respect to a vertex, we iteratively compute the corresponding upper bound of l simultaneously. Finally, we utilize the definition of D-core to iteratively refine all the upper bounds to obtain the anchored corenesses of all vertices.

Note that the anchored coreness-based algorithm first fixes one dimension and then computes the anchored corenesses for the other dimension, which may lead to suboptimal performance. To improve performance, we further propose a novel concept, called *skyline coreness*, and develop a *skyline coreness*-based algorithm. Specifically, we say the pair (k_v, l_v) is a skyline coreness of a vertex v, if there is no other pair (k'_v, l'_v) such that $k'_v \geq k_v$, $l'_v \geq l_v$, and $v \in (k'_v, l'_v)$-core. For example, in Figure 1, the skyline corenesses of v_2 are $\{(2, 2), (3, 1)\}$. Compared with anchored corenesses, a vertex's skyline corenesses contain fewer pairs of (k_v, l_v). Nevertheless, based on the skyline corenesses, we can still easily find all the D-cores containing the corresponding vertex. To be specific, if (k_v, l_v) is a skyline coreness of v, then v is also in the (k, l)-cores with $k \leq k_v$ and $l \leq l_v$. The basic idea of the skyline coreness-based algorithm is to use neighbors' skyline corenesses to iteratively estimate the skyline corenesses of each vertex. To this end, we define a new index, called *D-index*, for each vertex based on the following unique property of skyline corenesses. If (k_v, l_v) is one of the skyline corenesses of a vertex v, we have (i) v has at least k_v in-neighbors such that each of these in-neighbors, v_i, has a skyline coreness (k'_{v_i}, l'_{v_i}) satisfying $k'_{v_i} \geq k_v$ and $l'_{v_i} \geq l_v$; and (ii) v has at least l_v out-neighbors such that each of these out-neighbors, v_j, has a skyline coreness (k''_{v_j}, l''_{v_j}) satisfying $k''_{v_j} \geq k_v$ and $l''_{v_j} \geq l_v$. With this property, we design a distributed algorithm to iteratively compute the D-index for each vertex with its neighbors' D-indexes. To deal with the combinatorial blow-ups in the computation of D-indexes, we further develop three optimization strategies to improve efficiency.

We implement our algorithms under two well-known distributed graph processing frameworks, i.e., vertex-centric [1, 28, 30, 35] and block-centric [12, 41, 45]. Empirical results on small graphs demonstrate that our algorithms run faster than the peeling-based algorithm by up to 3 orders of magnitude. For larger graphs with more than 50 million edges, the peeling-based algorithm cannot finish within 5 days, while our algorithms can finish within 1 hour for most datasets. Moreover, our proposed algorithms require less than 100 rounds to converge for most datasets, and more than 90% vertices can converge within 10 rounds.

This paper's main contributions are summarized as follows:

- For the first time in the literature, we study the problem of distributed D-core decomposition over large directed graphs.
- We develop an anchored coreness-based distributed algorithm using well-defined in-H-index and out-H-index. To efficiently compute the anchored corenesses, we propose tight upper bounds that can be iteratively refined to exact anchored corenesses with reduced network communications.
- We further propose a novel concept of skyline coreness and show that the problem is equivalent to the computation of skyline corenesses for all vertices. A new two-dimensional D-index that unifies the in- and out-neighbor relationships, together with three optimization strategies, is designed to compute the skyline corenesses distributedly.
- Both theoretical analysis and empirical evaluation validate the efficiency of our algorithms for distributed D-core decomposition.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 formally defines the problem. Sections 4 and 5 propose two distributed algorithms for computing anchored coreness and skyline coreness, respectively. Experimental results are reported in Section 6. Finally, Section 7 concludes the paper.

2 RELATED WORK

In this section, we review the related work in two aspects, i.e., *core decomposition* and *distributed graph computation*.

Core Decomposition. As a well-known dense subgraph model, a k-core is the maximal subgraph of an undirected graph such that every vertex has at least k neighbors within this subgraph [39]. The core decomposition task aims at finding the k-cores for all possible values of k in a graph. Many efficient algorithms have been proposed to handle core decomposition over an undirected graph, such as peeling-based algorithms [4, 8, 22], disk-based algorithm [8, 22],

semi-external algorithm [43], streaming algorithms [36, 37], parallel algorithms [9, 11], and distributed algorithms [3, 31, 33]. It is worth mentioning that the distributed algorithms for k-core decomposition [3, 31, 33] cannot be used for distributed D-core decomposition. Specifically, the distributed k-core decomposition algorithms use the neighbors' corenesses to estimate a vertex's coreness, where all neighbors are of the same type. For D-core, a vertex's neighbors include in-neighbors and out-neighbors, which affect each other and should be considered simultaneously. If we consider only one type of neighbors, we cannot get the correct answer. Inspired by the H-index-based computation for core decomposition [29] and nucleus decomposition [38], we apply a similar idea in the design of distributed algorithms. Nevertheless, our technical novelty lies in the non-trivial extension of H-index from *one-dimensional undirected coreness* to *two-dimensional anchored/skyline coreness*, which needs to consider the computations of in-degrees and out-degrees simultaneously in a unified way.

In addition, core decomposition has been studied for different types of networks, such as weighted graphs [10, 46], uncertain graphs [5, 34], bipartite graphs [25], temporal graphs [15, 44], and heterogeneous information networks [14]. Recently, a new problem of distance-generalized core decomposition has been studied by considering vertices' h-hop connectivity [6, 27]. Note that a directed graph can be viewed as a bipartite graph. After transforming a directed graph to a bipartite graph, the (k, l)-core in the directed graph has a corresponding (α, β)-core in the bipartite graph [25], but not vice versa. Therefore, the problems of (k, l)-core decomposition and (α, β)-core decomposition are not equivalent, and (α, β)-core decomposition algorithms cannot be used in our work.

Distributed Graph Computation. In the literature, there exist various distributed graph computing models and systems to support big graph analytics. Among them, the *vertex-centric* framework [28, 30, 32] and the *block-centric* framework [41, 45] are two most popular frameworks.

The vertex-centric framework assumes that each vertex is associated with one computing node and communication occurs through edges. The workflow of the vertex-centric framework consists of a set of synchronous supersteps. Within each superstep, the vertices execute a user-defined function asynchronously after receiving messages from their neighbors. If a vertex does not receive any message, it will be marked as inactive. The framework stops once all vertices become inactive. Typical vertex-centric systems include Pregel [30], Giraph [1], GPS [35], and GraphLab [28]. For the block-centric framework, one computing node stores the vertices within a block together and communication occurs between blocks after the computation within a block reaches convergence. Compared with the vertex-centric framework, the block-centric framework can reduce the network traffic and better balance the workload among nodes. Distributed graph processing systems such as Giraph++ [41], Blogel [45], and GRAPE [12] belong to the block-centric framework.

Note that in this paper we mainly focus on algorithm designs for distributed D-core decomposition. To demonstrate the flexibility of our proposed algorithms, we implement them for performance evaluation in both vertex-centric and block-centric frameworks.

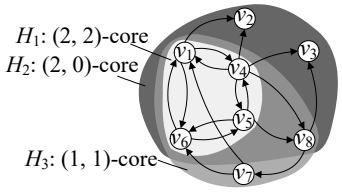


Figure 2: D-core

3 PROBLEM FORMULATION

In this paper, we consider a directed, unweighted simple graph $G = (V_G, E_G)$, where V_G and E_G are the set of vertices and edges, respectively. Each edge $e \in E_G$ has a direction. For an edge $e = \langle u, v \rangle \in E_G$, we say u is an in-neighbor of v and v is an out-neighbor of u. Correspondingly, $N_G^{in}(v)$ and $N_G^{out}(v)$ are respectively denoted as the in-neighbor set and out-neighbor set of a vertex v in G. We define three kinds of degrees for a vertex v as follows: (1) the in-degree is the number of v's in-neighbors in G, i.e., $deg_G^{in}(v) = |N_G^{in}(v)|$; (2) the out-degree is the number of v's out-neighbors in G, i.e., $deg_G^{out}(v) = |N_G^{out}(v)|$; (3) the degree is the sum of its in-degree and out-degree, i.e., $deg_G(v) = deg_G^{in}(v) + deg_G^{out}(v)$. Based on the in-degree and out-degree, we give a definition of D-core as follows.

Definition 3.1. ***D-core [17].*** *Given a directed graph $G = (V_G, E_G)$ and two integers k and l, a D-core of G, also denoted as (k, l)-core, is the maximal subgraph $H = (V_H, E_H) \subseteq G$ such that $\forall v \in V_H$, $deg_H^{in}(v) \geq k$ and $deg_H^{out}(v) \geq l$.*

According to Definition 3.1, a D-core should satisfy both the degree constraints and the size constraint. The degree constraints ensure the cohesiveness of D-core in terms of in-degree and out-degree. The size constraint guarantees the uniqueness of the D-core, i.e., for a specific pair of (k, l), there exists at most one D-core in G. Moreover, D-core has a *partial nesting* property as follows.

Property 3.1. ***Partial Nesting.*** *Given two D-cores, (k_1, l_1)-core D_1 and (k_2, l_2)-core D_2, D_1 is nested in D_2 (i.e., $D_1 \subseteq D_2$) if $k_1 \geq k_2$ and $l_1 \geq l_2$. Note that if $k_1 \geq k_2$ and $l_1 < l_2$, or $k_1 < k_2$ and $l_1 \geq l_2$, D_1 and D_2 may be not nested in each other.*

Example 3.1. *In Figure 2, the directed subgraph H_1 induced by the vertices v_1, v_4, v_5, and v_6 is a $(2, 2)$-core since $\forall v \in V_{H_1}$, $deg_{H_1}^{in}(v) = deg_{H_1}^{out}(v) = 2$. Moreover, $H_1 \subseteq H_2 = (2, 0)$-core, $H_1 \subseteq H_3 = (1, 1)$-core. On the other hand, $H_2 \not\subseteq H_3$ and $H_3 \not\subseteq H_2$, due to the non-overlapping vertices v_2, v_3, and v_7.*

In this paper, we study the problem of D-core decomposition to find all D-cores of a directed graph G in distributed settings. In-memory algorithms of D-core decomposition have been studied in [13, 17], assuming that the entire graph and associated structures can fit into the memory of a single machine. To our best knowledge, the problem of distributed D-core decomposition, considering a large graph distributed over a collection of machines, has not been investigated in the literature. We formulate a new problem of distributed D-core decomposition as follows.

Problem 1. **(Distributed D-core Decomposition).** *Given a directed graph $G = (V_G, E_G)$ that is distributed in a collection of machines $\{M_i : a\ machine\ M_i\ holds\ a\ partial\ subgraph\ G_i \subseteq G, 1 \leq i \leq n\}$ where $n \geq 2$ and $\cup_{i=1}^{n} G_i = G$, the problem of distributed D-core decomposition is to find all D-cores of G using n machines, i.e., identifying the (k, l)-cores with all possible (k, l) pairs.*

Consider applying D-core decomposition on G in Figure 2. We can obtain a total of 9 different D-cores: $(0,0)$-core = $(1,0)$-core = G; $(0,1)$-core = $(1,1)$-core = H_3; $(0,2)$-core = the subgraph of G induced by the vertices in $V_{H_1} \cup \{v_7\}$; $(1,2)$-core = $(2,1)$-core = $(2,2)$-core = H_1; $(2,0)$-core = H_2.

In the following two sections, we propose two new distributed algorithms for D-core decomposition. Without loss of generality, we mainly present the algorithms under the vertex-centric framework. At the end of Sections 4 and 5, we discuss how to extend our proposed algorithms to the block-centric framework.

4 DISTRIBUTED ANCHORED CORENESS-BASED ALGORITHM

In this section, we first give a definition of anchored coreness, which is useful for D-core decomposition. Then, we present a vertex-centric distributed algorithm for anchored coreness computation. Finally, we analyze the correctness and complexity of our proposed algorithm, and discuss its block-centric extension.

4.1 Anchored Coreness

Recall that, in the undirected k-core model [39], every vertex v has a unique value called *coreness*, i.e., the maximum value $k \in \mathbb{N}_0$ such that v is contained in a non-empty k-core. Similarly, we give a definition of *anchored coreness* for directed graphs as follows.

Definition 4.1. ***(Anchored Coreness).*** *Given a directed graph G and an integer k, the anchored coreness of a vertex $v \in V_G$ is a pair $(k, l_{max}(v,k))$, where $l_{max}(v,k) = \max_{l \in \mathbb{N}_0}\{l \mid \exists (k,l)\text{-core } H \subseteq G \wedge v \in V_H\}$. The* entire anchored corenesses *of the vertex v are defined as $\Phi(v) = \{(k', l_{max}(v,k')) \mid 0 \le k' \le k_{max}(v)\}$, where $k_{max}(v) = \max_{k'' \in \mathbb{N}_0}\{k'' \mid \exists (k'',0)\text{-core } H \wedge v \in V_H\}$.*

Different from the undirected coreness, the anchored coreness is a two-dimensional feature of in-degree and out-degree in directed graphs. For example, consider the graph G in Figure 1 and $k = 3$, the anchored coreness of vertex v_2 is $(3,1)$, as $l_{max}(v_2, 3) = 1$. Correspondingly, $\Phi(v_2) = \{(0,2),(1,2),(2,2),(3,1)\}$. The anchored corenesses can facilitate the distributed D-core decomposition as follows. According to Property 3.1, for a vertex v with the anchored coreness of (k,l), v belongs to any (k,l')-core with $l' \le l$. Hence, as long as we compute the anchored corenesses of v for each possible k, we can get all D-cores containing v. As a result, for a given directed graph G, the problem of D-core decomposition is equivalent to computing the entire anchored corenesses for every vertex $v \in V_G$, i.e., $\{\Phi(v) | v \in V_G\}$.

4.2 Distributed Anchored Coreness Computing

In this section, we present a distributed algorithm for computing the entire anchored corenesses for every vertex in G.

Overview. To handle the anchored coreness computation simultaneously in a distributed setting, we propose a distributed vertex-centric algorithm to compute all feasible anchored corenesses (k,l)'s for a vertex v. The general idea is to first identify the feasible range of $k \in [0, k_{max}(v)]$ by exploring $(k,0)$-cores and then refine an estimated upper bound of $l_{max}(v,k)$ to be exact for all possible values of k. The framework is outlined in Algorithm 1, which gives an overview of the anchored coreness updating procedure in three phases: 1) deriving $k_{max}(v)$; 2) computing the upper bound of $l_{max}(v,k)$ for each k; and 3) refining the upper bound to the exact anchored coreness $l_{max}(v,k)$. Note that in the second and third phases, the upper bound of $l_{max}(v,k)$ can be computed and refined in batch, instead of one by one sequentially, for different values of $k \in [0, k_{max}(v)]$.

Algorithm 1: Distributed Anchored Coreness Computation: routine executed by a vertex v

Input: directed graph G, vertex v
Output: anchored corenesses of vertex v
1 Compute $k_{max}(v)$ for vertex v using Algorithm 2;
2 Compute the upper bounds $l_{upp}(k,v)$ where $k \in [0, k_{max}]$, by invoking Algorithm 3;
3 Refine the upper bounds $l_{upp}(k,v)$ to anchored corenesses $l_{max}(k,v)$ using Algorithm 4;
4 **return** the entire anchored corenesses of v as $\Phi(v)$;

Phase I: Computing the in-degree limit $k_{max}(v)$. To compute $k_{max}(v)$, first, we introduce a concept of H-index [18]. Specifically, given a collection of integers S, the H-index of S is a maximum integer h such that S has at least h integer elements whose values are no less than h, denoted as $\mathcal{H}(S)$. For example, given $S = \{1,2,3,3,4,6\}$, H-index $\mathcal{H}(S) = 3$, as S has at least 3 elements whose values are no less than 3. Based on H-index, we give a new definition of *n-order in-H-index* for directed graph.

Definition 4.2. **(*n*-order in-H-index).** *Given a vertex v in G, the n-order in-H-index of v, denoted by $\mathrm{iH}_G^{(n)}(v)$, is defined as*

$$\mathrm{iH}_G^{(n)}(v) = \begin{cases} deg_G^{in}(v), & n = 0 \\ \mathcal{H}(I), & n > 0 \end{cases} \tag{1}$$

where the integer set $I = \{\mathrm{iH}_G^{(n-1)}(u) | u \in N_G^{in}(v)\}$.

Theorem 4.1 (Convergence).

$$k_{max}(v) = \lim_{n \to \infty} \mathrm{iH}_G^{(n)}(v) \tag{2}$$

Proof. Due to space limitation, we give a proof sketch here. The detailed proof can be found in [23]. First, we prove that $\mathrm{iH}_G^{(n)}(v)$ is non-increasing with the increase of order n. Thus, $\mathrm{iH}_G^{(n)}(v)$ finally converges to an integer when n is big enough. Then, we prove $k_{max}(v) \le \mathrm{iH}_{G'}^{(\infty)}(v) \le \mathrm{iH}_G^{(\infty)}(v)$, where $G' \subseteq G$ is a subgraph induced by the vertices v' with $k_{max}(v') \ge k_{max}(v)$. Also, we know $k_{max}(v) \ge \mathrm{iH}_G^{(\infty)}(v)$ by definition. Hence, $k_{max}(v) = \mathrm{iH}_G^{(\infty)}(v)$. □

According to Theorem 4.1, $\mathrm{iH}_G^{(n)}(v)$ finally converges to $k_{max}(v)$, based on which we present a distributed algorithm as shown in Algorithm 2 to compute $k_{max}(v)$. Algorithm 2 has an initialization step (lines 1-4), and two update procedures after receiving one message (lines 5-7) and all messages (lines 8-11). It first uses 0 to initialize the set I, which keeps the latest n-order in-H-indexes of v's in-neighbors (lines 1-2). Then, the algorithm sets the n-order in-H-index of v to its in-degree (line 3) and sends the message <v, iH(v)> to all its out-neighbors (line 4). When v receives a message

Table 1: An illustration of distributed D-core decomposition using Algorithm 1 on graph G in Figure 2.

		Vertices							
		v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
Phase I	$\text{iH}^{(0)}(v)$	3	2	2	2	2	3	1	2
	$\text{iH}^{(1)}(v)$	**2**	2	2	2	2	**2**	1	2
	$\text{iH}^{(2)}(v) = k_{max}(v)$	2	2	2	2	2	2	1	2
Phase II	$\forall k \in [0, k_{max}(v)], \text{oH}^{(0)}_{G[k]}(v)$	3; 3; 3	0; 0; 0	0; 0; 0	5; 5; 5	3; 3; 3	2; 2; 2	2; 2	2; 2; 2
	$\forall k \in [0, k_{max}(v)], \text{oH}^{(1)}_{G[k]}(v)$	**2; 2; 2**	0; 0; 0	0; 0; 0	**2; 2; 2**	**2; 2; 2**	2; 2; 2	2; 2	**1; 1; 0**
	$\forall k \in [0, k_{max}(v)], \text{oH}^{(2)}_{G[k]}(v)$	2; 2; 2	0; 0; 0	0; 0; 0	2; 2; 2	2; 2; 2	2; 2; 2	2; 2	1; 1; 0
Phase III	$\forall k \in [0, k_{max}(v)], l_{upp}(k, v)$	2; 2; 2	0; 0; 0	0; 0; 0	2; 2; 2	2; 2; 2	2; 2; 2	2; 2	1; 1; 0
	$\forall k \in [0, k_{max}(v)], l'_{upp}(k, v)$	2; 2; 2	0; 0; 0	0; 0; 0	2; 2; 2	2; 2; 2	2; 2; 2	**2; 1**	1; 1; 0
	$\forall k \in [0, k_{max}(v)], l_{max}(k, v)$	2; 2; 2	0; 0; 0	0; 0; 0	2; 2; 2	2; 2; 2	2; 2; 2	2; 1	1; 1; 0

Algorithm 2: Computing $k_{max}(v)$

Input: directed graph G, vertex v
Output: $k_{max}(v)$

Initializations
1 **for** *each* $v' \in N^{in}_G(v)$ **do**
2 $\quad I[v'] \leftarrow 0$;
3 $\text{iH}(v) \leftarrow deg^{in}_G(v)$;
4 Send message $\langle v, \text{iH}(v)\rangle$ to all out-neighbors of v;

On receiving message $\langle v', \text{iH}(v')\rangle$ ***from*** v***'s in-neighbor*** v'
5 $I[v'] \leftarrow \text{iH}(v')$;
6 **if** $\text{iH}(v') < \text{iH}(v)$ **then**
7 $\quad flag \leftarrow$ True

After receiving all messages
8 **if** $flag = True$ **then**
9 $\quad$ **if** $\mathcal{H}(I) < \text{iH}(v)$ **then**
10 $\quad\quad \text{iH}(v) \leftarrow \mathcal{H}(I); flag \leftarrow$ False;
11 $\quad\quad$ Send message $\langle v, \text{iH}(v)\rangle$ to all out-neighbors of v

When no vertex broadcasts messages
12 **return** $k_{max}(v) \leftarrow \text{iH}(v)$;

<v′, iH(v′)> from its in-neighbor v', the algorithm updates the n-order in-H-index of v' (line 5). If $\text{iH}(v') < \text{iH}(v)$, it means the n-order in-H-index of v may decrease. Thus, *flag* is set to *True* to indicate the re-computation of v's n-order in-H-index (line 7). After receiving all massages, if *flag* is *True*, Algorithm 2 updates v's n-order in-H-index $\text{iH}(v)$ and inform all its out-neighbors if $\text{iH}(v)$ decreases (lines 9-11). Algorithm 2 completes and returns $\text{iH}(v)$ as $k_{max}(v)$ when there is no vertex broadcasting messages (line 12).

Example 4.1. *We use the directed graph G in Figure 2 to illustrate Algorithm 2, whose calculation process is shown in Table 1. We take vertex* v_1 *as an example. First,* v_1*'s 0-order in-H-index is initialized with its in-degree, i.e.,* $\text{iH}^{(0)}_G(v_1) = 3$*. Then, Algorithm 2 iteratively computes* $\text{iH}^{(n)}_G(v_1)$*. After one iteration, the 1-order in-H-index of* v_1 *has converged to* $\mathcal{S}(\text{iH}^{(0)}_G(v_4), \text{iH}^{(0)}_G(v_6), \text{iH}^{(0)}_G(v_7)) = \mathcal{S}(2, 3, 1) = 2$*. Thus,* $k_{max}(v_1) = \text{iH}^{(2)}_G(v_1) = \text{iH}^{(1)}_G(v_1) = 2$.

Phase II: Computing the upper bounds of $l_{max}(k, v)$. In a distributed setting, the computation of $l_{max}(k, v)$ faces technical challenges. It is difficult to compute $l_{max}(k, v)$ by making use of only the "intermediate" neighborhood information. Because some vertices $u \in N_G(v)$ may become disqualified and thus be removed from the candidate set of $(k, l_{max}(k, v))$-core during the iteration process. Even worse, verifying the candidacy of u requires a large number of message exchanges between vertices. To address these issues, we design a novel upper bound for $l_{max}(k, v)$, denoted by $l_{upp}(k, v)$, which can be iteratively computed with "intermediate" corenesses to reduce communication costs. To start with, we give a new definition of n-order out-H-index, similar to Definition 4.2.

Definition 4.3. **(n-order out-H-index).** *Given a vertex v in G, the n-order out-H-index of v, denoted as* $\text{oH}^{(n)}_G(v)$*, is defined as*

$$\text{oH}^{(n)}_G(v) = \begin{cases} deg^{out}_G(v), & n = 0 \\ \mathcal{H}(O), & n > 0 \end{cases} \tag{3}$$

where $O = \{\text{oH}^{(n-1)}_G(u) | u \in N^{out}_G(v)\}$.

Based on $\text{oH}^{(n)}_G(v)$, we have the following theorem.

Theorem 4.2. *Given a vertex v in G and an integer $k \in [0, k_{max}(v)]$, let $G[k]$ be the subgraph of G induced by the vertices in $V_k = \{u \mid u \in V_G \wedge k_{max}(u) \geq k\}$. Then, it holds that*

$$l_{max}(k, v) \leq \lim_{n \to \infty} \text{oH}^{(n)}_{G[k]}(v). \tag{4}$$

Proof. Similar to Theorem 4.1, we can prove $\lim_{n \to \infty} \text{oH}^{(n)}_{G[k]}(v) = l'$ such that $v \in (0, l')$-core of $G[k]$ but $v \notin (0, l'+1)$-core of $G[k]$. Then, we have the following relationship for the D-cores of $G[k]$: $(k, l_{max}(k, v))$-core $\subseteq (0, l_{max}(k, v))$-core $\subseteq (0, l')$-core. According to the partial nesting property of D-core, $l' \geq l_{max}(k, v)$ holds. □

Theorem 4.2 indicates that $\lim_{n \to \infty} \text{oH}^{(n)}_{G[k]}(v)$ can be served as an upper bound of $l_{max}(k, v)$, i.e., $l_{upp}(k, v) = \lim_{n \to \infty} \text{oH}^{(n)}_{G[k]}(v)$. Thus, we can compute $l_{upp}(k, v)$ by iteratively calculating the n-order out-H-index of v in the directed subgraph $G[k]$. Moreover, to efficiently compute $l_{upp}(k, v)$ for all values $k \in [0, k_{max}(v)]$ in parallel, our distributed algorithm should send updating messages in batch and compute $l_{upp}(k, v)$ simultaneously.

Based on the above discussion, we propose a distributed algorithm for computing the upper bounds $l_{upp}(k, v)$. Algorithm 3 presents the detailed procedure. First, it initializes the n-order out-H-index of v for each possible value of k and sends them to v's in-neighbors (lines 1-5). When v receives a message from its out-neighbor v', v updates the n-order out-H-index of v' for subsequent calculation (lines 6-10). After receiving all messages, v updates its own n-order out-H-index for each possible value of k

Algorithm 3: Computing Upper Bounds $l_{upp}(k, v)$

Input: directed graph G, vertex v, $k_{max}(v)$
Output: the upper bounds $l_{upp}(k, v)$ for $k \in [0, k_{max}(v)]$

Initializations
1 **for** *each* $k \in [0, k_{max}(v)]$ **do**
2 **for** *each* $v' \in N^{out}_{G[k]}(v)$ **do**
3 $I[k][v'] \leftarrow 0$;
4 $\text{oH}_v[k] \leftarrow deg^{out}_{G[k]}(v)$; $change[k] \leftarrow$ True;
5 Send message $\langle v, \text{oH}_v[\cdot], change[\cdot]\rangle$ to all in-neighbors of v;

On receiving message $\langle v', \text{oH}_{v'}[\cdot], change[\cdot]\rangle$ ***from*** *v's* ***out-neighbor*** v'
6 **for** *each* $k \in [0, k_{max}(v)]$ **do**
7 **if** $change[k] = True$ **then**
8 $I[k][v'] \leftarrow \text{oH}[k]$;
9 **if** $\text{oH}_{v'}[k] < \text{oH}_v[k]$ **then**
10 $flag[k] \leftarrow$ True

After receiving all messages
11 **for** *each* $k \in [0, k_{max}(v)]$ **do**
12 **if** $flag[k] = True$ **then**
13 **if** $\mathcal{H}(I[k]) < \text{oH}_v[k]$ **then**
14 $\text{oH}_v[k] \leftarrow \mathcal{H}(I[k])$; $change[k] \leftarrow$ False;
15 **if** $\exists k \in [0, k_{max}(v)]$, $change[k] = True$ **then**
16 Send message $\langle v, \text{oH}_v[\cdot], change[\cdot]\rangle$ to all in-neighbors of v;

When no vertex broadcasts messages
17 $l_{upp}[\cdot] \leftarrow \text{oH}_v[\cdot]$;

Algorithm 4: Anchored Coreness Refinement

Input: graph G, vertex v, $k_{max}(v)$, upper bounds $l_{upp}[\cdot]$
Output: the entire anchored corenesses of v as $\Phi(v)$

Initializations
1 **for** *each* $k \in [0, k_{max}(v)]$ **do**
2 $change[k] \leftarrow$ True; $l[k][v] \leftarrow 0$;
3 Send message $\langle v, l_{upp}[\cdot], change[\cdot]\rangle$ to all neighbors of v;

On receiving message $\langle v', l_{upp}[\cdot], change[\cdot]\rangle$ ***from*** *v's* ***neighbor*** v'
4 **for** *each* $k \in [0, k_{max}(v)]$ **do**
5 **if** $change[k] = True$ **then**
6 $l[k][v'] \leftarrow l_{upp}[k]$;
7 $flag[k] \leftarrow$ True

After receiving all messages
8 **for** *each* $k \in [0, k_{max}(v)]$ **do**
9 **if** $flag[k] = True$ **then**
10 $V' \leftarrow \{v' | v' \in N^{in}_G(v) \wedge l[k][v'] \geq l_{upp}[k]\}$;
11 $V'' \leftarrow \{v'' | v'' \in N^{out}_G(v) \wedge l[k][v''] \geq l_{upp}[k]\}$;
12 **if** $|V'| < k$ ***or*** $|V''| < l_{upp}[k]$ **then**
13 $l_{upp}[k] \leftarrow l_{upp}[k] - 1$; $change[k] \leftarrow$ True;
14 **if** $\exists k \in [0, k_{max}(v)]$ *such that* $change[k] = True$ **then**
15 Send message $\langle v, l_{upp}[\cdot], change[\cdot]\rangle$ to all neighbors of v;

When no vertex broadcasts messages
16 **for** *each* $k \in [0, k_{max}(v)]$ **do**
17 Add $(k, l_{upp}[k])$ to the anchored corenesses $\Phi(v)$;

(lines 11-14). If any n-order out-H-indexes of v decreases, v informs all its in-neighbors (lines 15-16). Finally, when there is no vertex broadcasting messages, we get the upper bound $l_{upp}(k, v)$ for each $k \in [0, k_{max}(v)]$ (line 17).

Example 4.2. *We illustrate Algorithm 3 by continuing Example 4.1. As shown in Table 1, since* $k_{max}(v_1) = 2$, *we first initialize the* 0*-order out-H-indexes of* v_1 *as* $\text{oH}^{(0)}_{G[k]}(v_1) = 3$ *for each* $k \in \{0, 1, 2\}$. *After one iteration of computing the n-order out-H-indexes, all* 1*-order out-H-indexes of* v_1 *have converged to 2. Thus, we have* $\text{oH}^{(1)}_{G[0]}(v_1) = 2$, $\text{oH}^{(1)}_{G[1]}(v_1) = 2$, $\text{oH}^{(1)}_{G[2]}(v_1) = 2$.

Phase III: Refining $l_{upp}(k, v)$ to $l_{max}(k, v)$. Finally, we present the third phase of refining the upper bound $l_{upp}(k, v)$ to get the exact anchored coreness $l_{max}(k, v)$. To this end, we first present the following theorem.

Theorem 4.3. *Given a vertex* v *in* G *and an integer* k, *if* $(k, l_{upp}(k, v))$ *is an anchored coreness of* v, *it should satisfy two constraints on in-neighbors and out-neighbors:* ***(i)*** v *has at least* k *in-neighbors* v' *such that* $l_{upp}(k, v') \geq l_{upp}(k, v)$; *and* ***(ii)*** v *has at least* $l_{upp}(k, v)$ *out-neighbors* v'' *such that* $l_{upp}(k, v'') \geq l_{upp}(k, v)$.

Theorem 4.3 obviously holds, according to Def. 3.1 of D-core and the upper bound $l_{upp}(k, v) \geq l_{max}(k, v)$. Based on Theorem 4.3, we can refine $l_{upp}(k, v)$ decrementally by checking the upper bounds $l_{upp}(k, v')$'s of v's in- and out-neighbors. If v satisfies the above two constraints in Theorem 4.3, $l_{upp}(k, v)$ keeps unchanged; otherwise, $l_{upp}(k, v)$ decreases by 1 as the current $(k, l_{upp}(k, v))$ is not an anchored coreness of v. The above process needs to repeat for all vertices and all possible values of k, until none of $(k, l_{upp}(k, v))$ changes. Finally, we obtain all anchored corenesses $\{\Phi(v) | v \in V_G\}$.

Algorithm 4 outlines the procedure of the distributed refinement phase. First, the algorithm initializes some auxiliary structures and broadcast v's upper bound $l_{upp}(k, v)$ for each possible $k \in [0, k_{max}(v)]$ (lines 1-3). When it receives a message from v's neighbor v', the algorithm updates the upper bound set for v' (lines 4-7). After receiving all messages, the algorithm refines $l_{upp}(k, v)$ for each $k \in [0, k_{max}(v)]$ based on Theorem 4.3 (lines 8-13). If there exists such a $(k, l_{upp}(k, v))$ whose $l_{upp}(k, v)$ is decreased, the algorithm broadcasts the new upper bound set to v's neighbors (lines 14-15). As soon as there are no vertex broadcasting messages, Algorithm 4 terminates and we get all anchored corenesses of v (lines 16-17).

Example 4.3. *Continue Example 4.2 to illustrate Algorithm 4 in Phase III, which refines the upper bound* $l_{upp}(k, v_1)$ *to the exact* $l_{max}(k, v_1)$. *For* $k_{max}(v_1) = 3$ *and each* $k \in [0, k_{max}(v_1)]$, *Table 1 reports the final results* $l_{upp}(k, v_1) = l_{max}(k, v_1) = 2$. *Therefore, the entire anchored corenesses of* v_1 *are* $\Phi(v_1) = \{(0, 2), (1, 2), (2, 2)\}$.

4.3 Algorithm Analysis and Extension

Complexity analysis. We first analyze the time, space, message complexities of Algorithm 1. Let the edge size $|E_G| = m$, the maximum in-degree $\Delta_{in} = \max_{v \in V_G} deg^{in}_G(v)$, the maximum out-degree $\Delta_{out} = \max_{v \in V_G} deg^{out}_G(v)$, and the maximum degree $\Delta = \max_{v \in V_G} deg_G(v)$. In addition, let R_{AC-I}, R_{AC-II}, and R_{AC-III} be

the number of convergence rounds required by the three phases in Algorithm 1, respectively. Let be the total number of converge rounds in Algorithm 1 as $R_{AC} = R_{AC-I} + R_{AC-II} + R_{AC-III}$ and $R_{AC} \in O(\Delta)$. We have the following theorems (their detailed proofs can be found in [23]):

Theorem 4.4. ***(Time and Space Complexities)*** *Algorithm 1 takes $O(R_{AC} \cdot \Delta_{in} \cdot \Delta)$ time and $O(\Delta_{in} \cdot \Delta)$ space. The total time and space complexities for computing all vertices' corenesses are $O(R_{AC} \cdot \Delta_{in} \cdot m)$ and $O(\Delta_{in} \cdot m)$, respectively.*

Theorem 4.5. ***(Message Complexity)*** *The message complexity (i.e., the total number of times that a vertex send messages) of Algorithm 1 is $O(\Delta_{in} \cdot \Delta_{out} \cdot \Delta)$. The total message complexity for computing all vertices' corenesses is $O(\Delta_{in} \cdot \Delta_{out} \cdot m)$.*

Block-centric extension of Algorithm 1. We further discuss to extend the vertex-centric D-core decomposition in Algorithm 1 to the block-centric framework. The extension can be easily achieved by changing the update operation after receiving all messages. That is, instead of having Algorithms 2, 3 and 4 perform the update operation only once after receiving all messages, in the block-centric framework, the algorithms should update the H-indexes multiple times until the local block converges. For example, Algorithm 2 computes the n-order in-H-index of v only once in each round (lines 10-13). In contrast, the block-centric version should compute v's n-order in-H-index iteratively with v's in-neighbors, that are located in the same block as v, before broadcasting messages to other blocks to enter the next round. Note that in the worst case, for block-centric algorithms, every vertex converges within the block after computing the in-H-index/out-H-index only once, which is the same as vertex-centric algorithms. Therefore, the worse-case cost of block-centric algorithms is the same as that of vertex-centric algorithms.

5 DISTRIBUTED SKYLINE CORENESS-BASED ALGORITHM

In this section, we propose a novel concept of skyline coreness, which is more elegant than the anchored coreness. Then, we give a new definition of n-order D-index for computing skyline corenesses. Based on the D-index, we propose a distributed algorithm for skyline coreness computation to accomplish D-core decomposition.

5.1 Skyline Coreness

Motivation. The motivation for proposing another skyline coreness lies in an important observation that the anchored corenesses (k, l)'s may have redundancy. For example, in Figure 1, the vertex v_2 has four anchored corenesses, i.e., $\Phi(v_2) = \{(0, 2), (1, 2), (2, 2), (3, 1)\}$. According to D-core's partial nesting property, if $v_2 \in (2, 2)$-core, v_2 must also belong to $(0, 2)$-core and $(1, 2)$-core. Thus, it is sufficient and more efficient to keep the coreness of v_2 as $\{(2, 2), (3, 1)\}$, which uses (2, 2)-core to represent other two D-cores (0, 2)-core and (1, 2)-core. This elegant representation is termed as *skyline coreness*, which can facilitate space saving and fast computation of D-core decomposition. Based on the above observation, we formally define the dominance operation and skyline coreness as follows.

Definition 5.1. **(Dominance Operations).** *Given two coreness pairs (k, l) and (k', l'), we define two operations '$\prec$' and '$\preceq$' to compare them: (i) $(k', l') \prec (k, l)$ indicates that (k, l) dominates (k', l'), i.e., either $k' < k$, $l' \leq l$ hold or $k' \leq k$, $l' < l$ hold; and (ii) $(k', l') \preceq (k, l)$ represents that $k' \leq k$, $l' \leq l$ hold.*

Definition 5.2. **(Skyline Coreness).** *Given a vertex v in a directed graph G and a coreness pair (k, l), we say that (k, l) is a skyline coreness of v* iff *it satisfies that (i) $v \in (k, l)$-core; and (ii) there exist no other pair (k', l') such that $(k, l) \prec (k', l')$ and $v \in (k', l')$-core. We use $\mathrm{SC}(v)$ to denote the* entire skyline corenesses *of the vertex v, i.e., $\mathrm{SC}(v) = \{(k, l) \mid (k, l)$ is a skyline coreness of $v\}$.*

In other words, the skyline coreness of a vertex v is a non-dominated pair (k, l) whose corresponding (k, l)-core contains v. For instance, vertex v_2 has the skyline corenesses $\mathrm{SC}(v_2) = \{(2, 2), (3, 1)\}$ in Figure 1, reflecting that no other coreness (k, l) can dominate any skyline coreness in $\mathrm{SC}(v_2)$. According to D-core's partial nesting property, for a skyline coreness (k, l) of v, v is contained in the (k', l')-core with $(k', l') \prec (k, l)$. Therefore, if we compute all skyline corenesses $\mathrm{SC}(v)$ for a vertex v, we can find all D-cores the vertex v belonging to. As a result, the problem of D-core decomposition is equivalent to computing the entire skyline corenesses for every vertex in G, i.e., $\{\mathrm{SC}(v) | v \in V_G\}$.

Structural properties of skyline coreness. We analyze the structural properties of skyline coreness.

Property 5.1. *Let (k_v, l_v) be a skyline coreness of v, the following properties hold:*

(I) *There exist k_v in-neighbors $v' \in N_G^{in}(v)$ such that $(k_v, l_v) \preceq (k_{v'}, l_{v'})$, and also l_v out-neighbors $v'' \in N_G^{out}(v)$ such that $(k_v, l_v) \preceq (k_{v''}, l_{v''})$.*

(II) *Two cases **cannot** hold in either way: there exist $k_v + 1$ in-neighbors $v' \in N_G^{in}(v)$ such that $(k_v+1, l_v) \preceq (k_{v'}, l_{v'})$, or l_v out-neighbors $v'' \in N_G^{out}(v)$ such that $(k_v+1, l_v) \preceq (k_{v''}, l_{v''})$.*

(III) *Two cases **cannot** hold in either way: there exist k_v in-neighbors $v' \in N_G^{in}(v)$ such that $(k_v, l_v + 1) \preceq (k_{v'}, l_{v'})$, or $l_v + 1$ out-neighbors $v'' \in N_G^{out}(v)$ such that $(k_v, l_v + 1) \preceq (k_{v''}, l_{v''})$.*

Proof. First, we prove Property 5.1(I). Let D_1 be the (k_v, l_v)-core of G, we have $deg_{D_1}^{in}(v) \geq k_v$ and $deg_{D_1}^{out}(v) \geq l_v$. For $\forall v' \in (N_{D_1}^{in}(v) \cup N_{D_1}^{out}(v))$, v' may be in the (k', l')-core with $k_v \leq k' \leq k_{v'}$ and $l_v \leq l' \leq l_{v'}$. Therefore, (I) of Property 5.1 holds.

Next, we prove Property 5.1(II). Assume that v has $k_v + 1$ in-neighbors V' and l_v out-neighbors V'' satisfying the constraints of (II). Then, V' and V'' must be in the $(k_v + 1, l_v)$-core. Moreover, $v \cup (k_v + 1, l_v)$-core is also a $(k_v + 1, l_v)$-core. Hence, $(k_v + 1, l_v)$ rather than (k_v, l_v) is a skyline coreness of v, which contradicts to the condition of Property 5.1. Therefore, the assumption does not hold.

Finally, Property 5.1(III) can be proved in the same way of Property 5.1(II). It is omitted due to space limitation. □

For example, $(2, 2)$ is a skyline coreness of v_2 in Figure 1. The in-neighbors of v_2 are v_3, v_4, v_5, and v_7, whose skyline corenesses are $\{(3, 3)\}$, $\{(2, 2)\}$, $\{(3, 3)\}$, and $\{(2, 2), (3, 1)\}$, respectively. These four vertices all have skyline corenesses that dominate or are identical to v_2's skyline coreness $(2, 2)$. But only two vertices v_3 and v_5 have skyline corenesses that dominate $(k_{v_2} + 1, l_{v_2}) = (3, 2)$. Hence,

$(3,2)$ is not a skyline coreness of v_2. Property 5.1 reveals the relationships among vertices' skyline corenesses, based on which we propose an algorithm for skyline coreness computation in the next subsection.

5.2 Distributed Skyline Corenesses Computing

We begin with a novel concept of D-index.

Definition 5.3. **(D-index).** *Given two sets of pairs of integers R_{in}, $R_{out} \subseteq \mathbb{N}_0 \times \mathbb{N}_0$, the D-index of R_{in} and R_{out} is denoted by $\mathcal{D}(R_{in}, R_{out}) \subseteq \mathbb{N}_0 \times \mathbb{N}_0$, where each element $(k,l) \in \mathcal{D}(R_{in}, R_{out})$ satisfies: (i) there exist at least k pairs $(k_i, l_i) \in R_{in}$ such that $(k,l) \preceq (k_i, l_i)$ for $1 \le i \le k$; (ii) there exist at least l pairs $(k_j, l_j) \in R_{out}$ such that $(k,l) \preceq (k_j, l_j)$ for $1 \le j \le l$; (iii) there does not exist another $(k', l') \in \mathbb{N}_0 \times \mathbb{N}_0$ satisfying the above conditions (1) and (2), and $(k,l) \prec (k', l')$.*

The idea of D-index is very similar to H-index. Actually, the D-index is an extension of H-index to handle two-dimensional integer pairs. For $\mathcal{D}(R_{in}, R_{out})$, it finds a series of (k,l) skyline pairs such that each has at least k dominated pairs in R_{in} and at least l dominated pairs in R_{out}, using a joint indexing way. For example, let $R_{in} = \{(1,1),(2,2)\}$ and $R_{out} = \{(3,3),(4,4)\}$, then $\mathcal{D}(R_{in}, R_{out}) = \{(1,2)\}$. Note that $\mathcal{D}(R_{in}, R_{out}) \neq \mathcal{D}(R_{out}, R_{in})$ may hold for the D-index, as $\mathcal{D}(R_{out}, R_{in}) = \{(2,1)\} \neq \{(1,2)\} = \mathcal{D}(R_{in}, R_{out})$ in this example. Next, we introduce another concept of n-order D-index for distributed D-core decomposition.

Definition 5.4. **(n-order D-index).** *Given a vertex v in G, the n-order D-index of v, denoted by $D^{(n)}(v) \subseteq \mathbb{N}_0 \times \mathbb{N}_0$, is defined as*

$$D^{(n)}(v) = \begin{cases} \{(deg_G^{in}(v), deg_G^{out}(v))\}, & n = 0 \\ \mathcal{D}(R_{in}^{(n-1)}(v), R_{out}^{(n-1)}(v)), & n > 0 \end{cases} \tag{5}$$

Here, $R_{in}^{(n-1)}(v) = \{(k_u, l_u) \in D^{(n-1)}(u) \mid u \in N_G^{in}(v)\}$ and $R_{out}^{(n-1)}(v) = \{(k_u, l_u) \in D^{(n-1)}(u) \mid u \in N_G^{out}(v)\}$. Note that $D^{(n)}(v)$ is the largest non-dominated *D-index such that it dominates or at least is identical to $\mathcal{D}(R_{in}^{(n-1)}(v), R_{out}^{(n-1)}(v))$, for each $R_{in}^{(n-1)}(v) \in D^{(n-1)}(u_1) \times \ldots \times D^{(n-1)}(u_i)$ when $N_G^{in}(v) = \{u_1, \ldots, u_i\}$ and each $R_{out}^{(n-1)}(v) \in D^{(n-1)}(u_1) \times \ldots \times D^{(n-1)}(u_j)$ when $N_G^{out}(v) = \{u_1, \ldots, u_j\}$.*

The n-order D-index $D^{(n)}(v)$ may contain more than one pair (k,l), i.e., $|D^{(n)}(v)| \ge 1$. Note that $R_{in}^{(n-1)}(v)$ and $R_{out}^{(n-1)}(v)$ consist of one pair (k_u, l_u) for each in-neighbor $u \in N_G^{in}(v)$ and each out-neighbor $u \in N_G^{out}(v)$, respectively. Therefore, there exist multiple combinations of $R_{out}^{(n-1)}(v)$ and $R_{out}^{(n-1)}(v)$. Moreover, $D^{(n)}(v)$ should consider all combinations of $R_{out}^{(n-1)}(v)$ and $R_{out}^{(n-1)}(v)$, and finally select the "best" choice as the *largest non-dominated* set of D-index $\mathcal{D}(R_{in}^{(n-1)}(v), R_{out}^{(n-1)}(v))$.

For two pair sets $R_1, R_2 \subseteq \mathbb{N}_0 \times \mathbb{N}_0$, we say $R_2 \preceq R_1$ if and only if $\forall (k,l) \in R_2, \exists (k', l') \in R_1$ such that $(k,l) \preceq (k', l')$. Then, we have the following theorem of n-order D-index convergence.

Theorem 5.1 (n-order D-index Convergence). *For a vertex v in G, it holds that*

$$\mathrm{SC}(v) = \lim_{n \to \infty} D^{(n)}(v) \tag{6}$$

Proof. The proof can be similarly done as Theorem 4.1. □

By Theorem 5.1, we can compute vertices' skyline corenesses via iteratively computing their n-order D-indexes until convergence.

5.3 Algorithms and Optimizations

A naive implementation of the distributed algorithm to compute $D^{(n)}(v)$ may suffer from serious performance problems, due to the combinatorial blow-ups in a large number of choices of $R_{in}^{(n-1)}(v)$ and $R_{out}^{(n-1)}(v)$. Thus, we first tackle three critical issues for fast distributed computation of n-order D-index.

Optimization-1: Fast computation of D-index $\mathcal{D}(R_{in}, R_{out})$. The first issue is, given R_{in} and R_{out}, how to compute D-index $\mathcal{D}(R_{in}, R_{out})$. A straightforward way is to list all candidate pairs and return the pairs satisfying Def. 5.3. According to conditions (1)&(2) in Def. 5.3, if (k,l) belongs to D-index, there exists at least k pairs of R_{in} satisfying the dominance relationship. Therefore, $0 \le k \le |R_{in}|$. Similarly, $0 \le l \le |R_{out}|$. Thus, there are a total of $(|R_1|+1) \cdot (|R_2|+1)$ candidate pairs to be checked, which is costly for large $|R_1|$ and $|R_2|$. In addition, the basic operation of D-index computation is frequently invoked in the process of computing $D^{(n)}(v)$. Hence, it is necessary to develop faster algorithms. To this end, we try to reduce the pairs for examination as many as possible through the following two optimizations.

- *Reducing the ranges of k and l.* For conditions (1)&(2) in Def. 5.3, if (k,l) belongs to $\mathcal{D}(R_{in}, R_{out})$, there exist at least k pairs (k_i, l_i) in R_{in} such that $(k,l) \preceq (k_i, l_i)$. In other words, at least k pairs (k_i, l_i) in R_{in} have $k_i \ge k$. Thus, the maximum k is denoted by $k_{max} = \mathcal{H}(I_k)$, where $I_k = \{k_i \mid (k_i, l_i) \in R_{in}\}$. Similarly, we can also obtain the maximum l, denoted by l_{max}, as $l_{max} = \mathcal{H}(O_l)$, where $O_l = \{l_j \mid (k_j, l_j) \in R_{out}\}$. Since $\mathcal{H}(I_k) \le |R_{in}|$ and $\mathcal{H}(O_l) \le |R_{out}|$, the total number of candidate pairs decreases.
- *Pruning disqualified candidate pairs.* Let $(k,l) \in \mathcal{D}(R_{in}, R_{out})$. According to condition (3) in Def. 5.3, if $(k', l') \in \mathcal{D}(R_{in}, R_{out})$ with $k' < k$, l' must satisfy $l' > l$. Otherwise, $(k', l') \prec (k,l)$ and $(k', l') \notin \mathcal{D}(R_{in}, R_{out})$. This rule can be used to prune disqualified pairs based on the found skyline corenesses.

Optimization-2: Fast computation of n-order D-index $D^{(n)}(v)$. The second issue is the computation of $D^{(n)}(v)$. By Def. 5.4, both $R_{in}^{(n-1)}(v)$ and $R_{out}^{(n-1)}(v)$ may have multiple instances. Hence, a straightforward way is to compute the D-index for every instance and finally integrate them together. In total, we need to compute the D-index $O(\prod_{v' \in N_G^{in}(v)} |D^{(n-1)}(v')| \cdot \prod_{v'' \in N_G^{out}(v)} |D^{(n-1)}(v'')|)$ times, which is very inefficient. Actually, several redundant computations occur due to many independent instances in the D-index computation. For example, in one instance, we have verified that (k,l) belongs to the n-order D-index. Then, there is no need to verify (k,l) in other instances. This motivates us to devise a more efficient method to compute $D^{(n)}(v)$, which requires D-index computation only once. Specifically, we first compute k_{max} and l_{max}. Then, we enumerate candidate pairs for dominance checking. Here,

Algorithm 5: Distributed Skyline Corenesses Computation Algorithm: routine executed by vertex v

Input: directed graph G, vertex v
Output: the skyline corenesses $\mathrm{SC}(v)$

Initializations
1 Compute $\mathrm{iH}_G^{(\infty)}(v)$ and $\mathrm{oH}_G^{(\infty)}(v)$ using Algorithm 2;
2 $D_v = \{(\mathrm{iH}_G^{(\infty)}(v), \mathrm{oH}_G^{(\infty)}(v))\}$;
3 Send message $\langle v, D_v \rangle$ to all neighbors of v;

On receiving message $\langle v', D_{v'} \rangle$ ***from*** v***'s neighbor*** v'
4 $D_k[v'] \leftarrow 0$; $D_l[v'] \leftarrow 0$;
5 $D[v'] \leftarrow D_{v'}$;
6 **for** *each* $(k,l) \in D_{v'}$ **do**
7 $\quad D_k[v'] \leftarrow \max(D_k[v'], k)$; $D_l[v'] \leftarrow \max(D_l[v'], l)$;
8 $flag \leftarrow$ True;

After receiving all messages
9 **if** $flag = True$ **then**
10 $\quad$ Apply Algorithm 6 on n-order D-index computation;
11 $\quad$ **if** $D[v] \neq D$ **then**
12 $\quad\quad D[v] \leftarrow D$; $D_v \leftarrow D$;
13 $\quad\quad$ Send message $\langle v, D_v \rangle$ to all neighbors of v;

When no vertex broadcasts messages
14 **return** $\mathrm{SC}(v) \leftarrow D[v]$;

we highlight two differences from the original D-index computation method.

- *The difference of k_{max} and l_{max} computations.* For k_{max} and l_{max} in D-index computation, I_k (resp. O_l) is formed by just adding k_i (resp. l_i) from each pair in R_{in}. For n-order D-index computation, the vertex's $(n-1)$-order D-index may have more than one pairs. We should select the maximum k_i and l_i among these pairs. Specifically, for v's n-order D-index computation, to compute k_{max}, $I_k(v) = \{k_i \mid v' \in N_G^{in}(v), k_i = \max_{(k_i', l_i') \in D^{(n-1)}(v')}(k_i')\}$. In the same way, $I_l(v) = \{l_j \mid v' \in N_G^{out}(v), l_j = \max_{(k_j', l_j') \in D^{(n-1)}(v')}(l_j')\}$.
- *The difference of dominance checking.* For a candidate pair (k,l), the D-index computation should find the pairs in R_{in} and R_{out} that dominate or are identical to (k,l). To compute $D^{(n)}(v)$, we should find all v's neighbors v' whose $(n-1)$-order D-index has a pair dominating or identical to (k,l). If $D^{(n-1)}(v')$ has multiple pairs, we need to examine the dominance relationship for each of these pairs with (k,l). Once one pair dominates or is identical to (k,l), such v' is identified.

Optimization-3: Tight initialization. Finally, we present an optimization for $D^{(n)}(v)$ computation using a tight initialization. In Def. 5.4, the 0-order D-index is initialized with the vertex's in-degree and out-degree. The optimization idea is that if we tightly initialize the vertex's 0-order D-index with smaller values (denoted by $D^{(0)}(v) = (k_0(v), l_0(v))$), the n-order D-index can converge faster to the exact skyline coreness. Here, we highlight two principles to find such $(k_0(v), l_0(v))$: (i) $k_0(v) \leq \max_{(k_i,l_i) \in \mathrm{SC}(v)} k_i$ and $l_0(v) \leq \max_{(k_i,l_i) \in \mathrm{SC}(v)} l_i$, otherwise the $D^{(n)}(v)$ cannot converge to $\mathrm{SC}(v)$; (ii) $(k_0(v), l_0(v))$ should be easy to compute in distributed settings. As a result, we present the following theorem.

Algorithm 6: n-order D-index Computation

Output: v's n-order D-index
1 $D \leftarrow \emptyset$; $l_{min} \leftarrow 0$;
2 $I_k = \{D_k[v'] | v' \in N_G^{in}(v)\}$; $k_{max} \leftarrow \mathcal{H}(I_k)$;
3 $O_l = \{D_l[v'] | v' \in N_G^{out}(v)\}$; $l_{max} \leftarrow \mathcal{H}(O_l)$;
4 **for** $k \leftarrow k_{max}$ **to** 0 **do**
5 $\quad l \leftarrow l_{max}$;
6 $\quad$ **while** $l > l_{min}$ **do**
7 $\quad\quad V_1 = \{v' | v' \in N_G^{in}(v)$, and $\exists (k', l') \in D[v']$, $(k,l) \preceq (k', l')\}$;
8 $\quad\quad V_2 = \{v' | v' \in N_G^{out}(v)$, and $\exists (k', l') \in D[v']$, $(k,l) \preceq (k', l')\}$;
9 $\quad\quad$ **if** $|V_1| \geq k \wedge |V_2| \geq l$ **then**
10 $\quad\quad\quad l_{min} \leftarrow l$; $D \leftarrow D \cup (k,l)$;
11 $\quad\quad l \leftarrow l - 1$;

Theorem 5.2. *For any vertex v in G, it holds that $k_{max}(v) \geq \max\{k_i \mid (k_i, l_i) \in \mathrm{SC}(v)\}$ and $l_{max}(v) \geq \max\{l_i \mid (k_i, l_i) \in \mathrm{SC}(v)\}$, where $l_{max}(v) = \max\{l \mid v \in (0,l)\text{-core} \wedge v \notin (0, l+1)\text{-core}\}$.*

Theorem 5.2 offers two tight upper bounds for k_0 and l_0, i.e., $k_{max}(v)$ and $l_{max}(v)$, respectively. In addition, according to Theorems 4.1 and 4.2, $k_{max}(v)$ and $l_{max}(v)$ can be computed by iteratively computing v's n-order in-H-index and out-H-index, respectively. Therefore, we initialize $D^{(0)}(v) = (k_{max}(v), l_{max}(v))$.

Algorithms. Based on the above theoretical analytics and optimizations, we present the distributed skyline corenesses computation algorithm in Algorithm 5. At the initialization phase, the algorithm computes $\mathrm{iH}_G^{(\infty)}(v)$ and $\mathrm{oH}_G^{(\infty)}(v)$ using Algorithm 2 and uses them to initialize the 0-order D-index of v, which is broadcast to all neighbors of v (lines 1-3). When v receives a message from its neighbor v', Algorithm 5 updates the n-order D-index of v' that is stored in v's node, and finds the maximum values in each pair of k and l (lines 4-8). After v receives all messages, Algorithm 5 computes the n-order D-index for v, which is described in Algorithm 6. Then, it broadcasts to all neighbors of v if the n-order D-index changes (lines 9-13). When there is no vertex broadcasting messages, Algorithm 5 returns the latest n-order D-index as skyline corenesses (line 14).

Next, we present the procedure of Algorithm 6 for n-order D-index computation. It first computes k_{max} and l_{max} as shown in Optimization-1 and Optimization-2 (lines 2-3), which help to determine the range of candidate pairs. Then, the algorithm enumerates all candidate pairs (k,l) and examines whether (k,l) belongs to the n-order D-index of v (lines 6-11). Note that l_{min} keeps the minimal value of l for the remaining candidate pairs, which is used to prune disqualified pairs.

Example 5.1. *We use the graph G in Figure 2 to illustrate Algorithm 5. Table 2 reports the process of computing skyline corenesses. Take vertex v_7 as an example. First, the 0-order D-index of v_7 is initialized with $\{(1,2)\}$, i.e., $D^{(0)}(v_7) = \{(1,2)\}$. Then, we iteratively compute the n-order D-index for v_7. We can observe that after one iteration only, the 1-order D-index of v_7 has converged as $D^{(2)}(v_7) = D^{(1)}(v_7) = \{(0,2),(1,1)\}$. Thus, the entire skyline corenesses of v_7 are $\mathrm{SC}(v_7) = \{(0,2),(1,1)\}$.*

Table 2: An illustration of distributed skyline coreness computation using Algorithm 5 on graph G in Figure 2.

	Vertices							
	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
$D^{(0)}(v)$	$\{(2,2)\}$	$\{(2,0)\}$	$\{(2,0)\}$	$\{(2,2)\}$	$\{(2,2)\}$	$\{(2,2)\}$	$\{(1,2)\}$	$\{(2,1)\}$
$D^{(1)}(v)$	$\{(2,2)\}$	$\{(2,0)\}$	$\{(2,0)\}$	$\{(2,2)\}$	$\{(2,2)\}$	$\{(2,2)\}$	$\{(0,2),(1,1)\}$	$\{(1,1),(2,0)\}$
$D^{(2)}(v)$	$\{(2,2)\}$	$\{(2,0)\}$	$\{(2,0)\}$	$\{(2,2)\}$	$\{(2,2)\}$	$\{(2,2)\}$	$\{(0,2),(1,1)\}$	$\{(1,1),(2,0)\}$

5.4 Algorithm Analysis and Extension

Complexity analysis. Let R_{SC} be the number of convergence rounds taken by Algorithm 5. In practice, our algorithms achieve $R_{SC} \leq R_{AC} \ll \Delta$ on real datasets. We show the time, space, and message complexities of Algorithm 5 below.

Theorem 5.3. *(**Time and Space Complexities**) Algorithm 5 takes $O(R_{SC} \cdot \Delta_{in} \cdot \Delta_{out})$ time and $O(\Delta \cdot \min\{\Delta_{in}, \Delta_{out}\})$ space. The total time and space complexities for computing all vertices' corenesses are $O(R_{SC} \cdot \Delta_{in} \cdot m)$ and $O(\min\{\Delta_{in}, \Delta_{out}\} \cdot m)$, respectively.*

Theorem 5.4. *(**Message Complexity**) The message complexity of Algorithm 5 is $O(\Delta^2)$. The total message complexity for computing all vertices' corenesses is $O(\Delta \cdot m)$.*

Through the above analysis, we can see that the skyline coreness-based approach in Algorithm 5 *takes less space* and *runs much faster* than the anchored coreness approach in Algorithm 1.

Block-centric extension. Algorithm 5 can be easily extended to the block-centric framework. The only difference is that each machine iteratively computes the n-order D-index locally until the algorithm converges within the local block, before broadcasting to other blocks (lines 9-13 of Algorithm 5).

6 PERFORMANCE EVALUATION

In this section, we empirically evaluate our proposed algorithms. We conduct our experiments on a collection of Amazon EC2 r5.2x large instances, each powered by 8 vCPUs and 64GB memory. The network bandwidth is up to 10G Gb/s. All experiments are implemented in C++ on the Ubuntu 18.04 operating system.

Datasets. We use 11 real-world graphs in our experiments. Table 3 shows the statistics of these graphs. Specifically, `Wiki-vote`[1] is a voting graph; `Email-EuAll`[1] is a communication graph; `Amazon`[1] is a product co-purchasing graph; `Hollywood`[2] is an actors collaboration graph; `Pokec`[1], `Live Journal`[1], and `Slashdot`[1] are social graphs; `Citation`[1] is a citation graph; `UK-2002`[2], `IT-2004`[2], and `UK-2005`[2] are web graphs.

Algorithms. We compare five algorithms in our experiments.

- **AC-V** and **AC-B**: The distributed anchored coreness-based D-core decomposition algorithms implemented in the vertex-centric and block-centric frameworks, respectively.
- **SC-V** and **SC-B**: The distributed skyline coreness-based D-core decomposition algorithms implemented in the vertex-centric and block-centric frameworks, respectively.
- **Peeling**: The distributed version of the peeling algorithm for D-core decomposition [13], in which one machine is assigned as the coordinator to collect global graph information and dispatch decomposition tasks.

[1]http://snap.stanford.edu/data/index.html
[2]http://law.di.unimi.it/datasets.php

Table 3: Statistics of the datasets (deg_{avg} represents the average degree; K = 10^3, M = 10^6, and B = 10^9)

Dataset	Abbr.	$\|V_G\|$	$\|E_G\|$	deg_{avg}	k_{max}	l_{max}
Wiki-vote	WV	7.1**K**	103.6**K**	14.57	19	15
Email-EuAll	EE	265.2**K**	420**K**	1.58	28	28
Slashdot	SL	82.1**K**	948.4**K**	11.54	54	9
Amazon	AM	400.7**K**	3.2**M**	7.99	10	10
Citation	CT	3.7**M**	16.5**M**	4.37	1	1
Pokec	PO	1.6**M**	30.6**M**	18.75	32	31
Live Journal	LJ	4.8**M**	69.0**M**	14.23	253	254
Hollywood	HW	2.1**M**	228.9**M**	105.00	1,297	99
UK-2002	UK2	18.5**M**	298.1**M**	16.09	942	99
UK-2005	UK5	39.4**M**	936.3**M**	23.73	584	99
IT-2004	IT	41.2**M**	1.1**B**	27.87	3,198	990

We employ GRAPE [12] as the block-centric framework and use the hash partitioner for graph partitioning by default. For the sake of fairness, we also employ GRAPE to simulate the vertex-centric framework. In specific, at each round, all vertices within a block execute computations only once and when all vertices complete the computation, the messages will be broadcast to their neighbors.

Parameters and Metrics. The parameters tested in experiments include # machines and graph size, whose default settings are 8 and $100\% \cdot |V_G|$, respectively. The performance metrics evaluated include # iterations required for convergence, convergence rate (i.e., the percentage of vertices who have computed the coreness), running time (in seconds), and communication overhead (i.e., the total messages sent by all vertices).

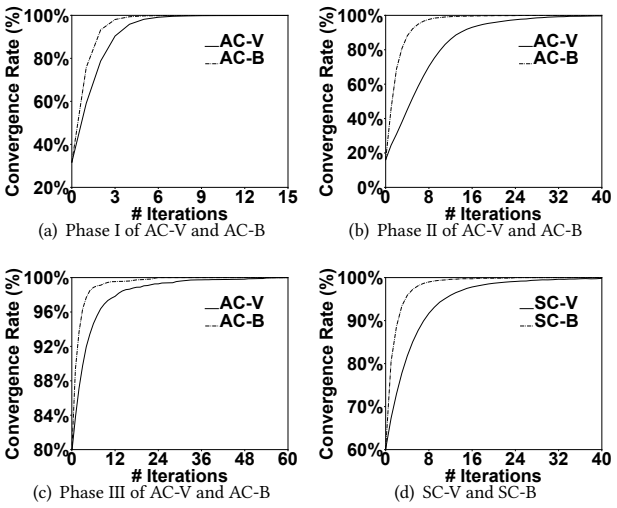
6.1 Convergence Evaluation

The first set of experiments evaluates the convergence of our proposed algorithms.

Exp-1: Evaluation on the number of iterations. We start by evaluating # iterations required for our algorithms to converge. Note that an iteration here refers to a cycle of the algorithm receiving messages, performing computations, and broadcasting messages. Table 4 reports the results on datasets `WV`, `EE`, `SL`, `AM`, and `CT`. We make several observations. First, for every graph, all of our proposed algorithms have much less iterations than the upper bound (i.e., the maximum degree of the graph), which demonstrates the efficiency of our algorithms. Second, the iterations of SC-V and SC-B are less than those of AC-V and AC-B. This is because the computation of anchored corenesses is more cumbersome than that of skyline corenesses. Hence, both AC-V and AC-B take more iterations. Third, for the same type of algorithms, i.e., AC or SC, the

Table 4: # Iterations required for the algorithms

Algorithms		Datasets				
		WV	EE	SL	AM	CT
Upper Bound		1,167	7,636	5,064	2,757	793
AC-V	Phase I	19	17	40	16	32
	Phase II	32	19	53	64	32
	Phase III	33	22	61	61	2
	Total	84	58	154	141	66
AC-B	Phase I	14	14	35	13	28
	Phase II	15	7	43	30	28
	Phase III	16	21	45	25	2
	Total	45	42	123	68	58
SC-V		33	19	61	65	**2**
SC-B		**17**	**6**	**46**	**25**	**2**



(a) Phase I of AC-V and AC-B (b) Phase II of AC-V and AC-B

(c) Phase III of AC-V and AC-B (d) SC-V and SC-B

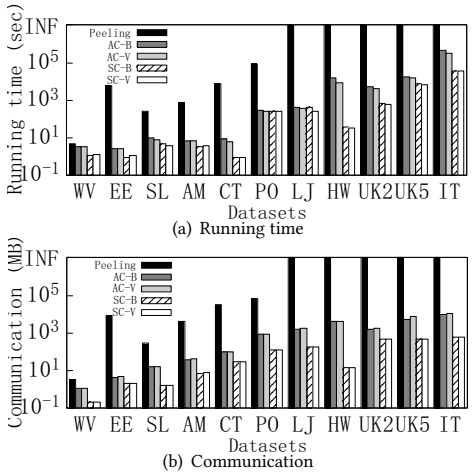
Figure 3: Convergence rates of our algorithms (AM)

algorithm implemented in the block-centric framework takes less iterations than that in the vertex-centric framework. The reason is that the block-centric framework allows algorithms to use vertices located in the same block to converge locally within a single round, which leads to faster convergence.

Exp-2: Evaluation on the convergence rate. Since different vertices require different numbers of iterations to converge, in this experiment, we evaluate the algorithms' convergence rates. Figure 3 shows the results on Amazon. As expected, the algorithms implemented in the block-centric framework converge faster. For example, in Figure 3(d), after 8 iterations, the convergence rates of SC-V and SC-B reach 89.9% and 98.6%, respectively. Moreover, most vertices can converge within just a few iterations. Specifically, for SC-B, more than 95% vertices converge within 5 iterations. In addition, SC algorithms have faster convergence rates than AC algorithms. For example, AC-B takes 68 iterations to reach convergence while SC-B takes 25 iterations.

6.2 Efficiency Evaluation

Next, we evaluate the efficiency of our proposed algorithms against the state-of-the-art peeling algorithm, denoted as Peeling. Note that if an algorithm cannot finish within 5 days, we denote it by 'INF'.



(a) Running time

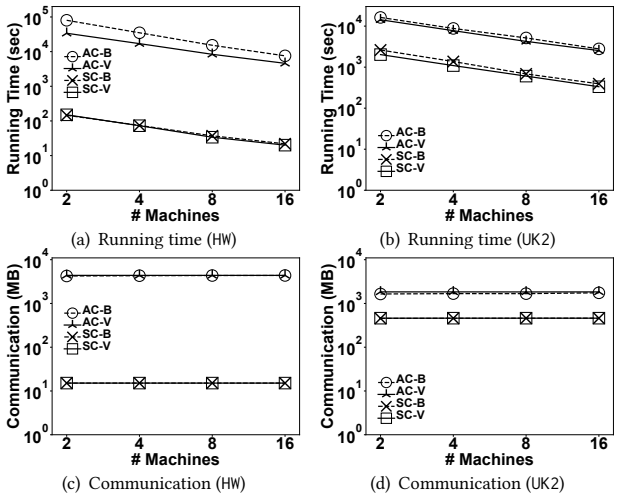
(b) Communication

Figure 4: Performance comparisons

Exp-3: Our algorithms vs. Peeling. We first compare the performance of our proposed algorithms with Peeling under the default experiment settings. Figures 4(a) and 4(b) report the results in terms of the running time and communication overhead, respectively. First, we can see that Peeling cannot finish within 5 days on the large-scale graphs with more than 50 million edges, including LJ, HW, UK2 UK5, and IT, while our algorithms can finish within 1 hour for most of these datasets and no more than 10 hours on the largest billion-scale graph for our fastest algorithm. Moreover, for the datasets where Peeling can finish, our algorithms outperform Peeling by up to 3 orders of magnitude. This well demonstrates the efficiency of our proposed algorithms. Second, SC-V and SC-B perform better than AC-V and AC-B in terms of both the running time and communication overhead. For example, on the biggest graph IT with over a billion edges, the improvement is nearly 1 order of magnitude. This is because SC-V and SC-B compute less corenesses than AC-V and AC-B. Third, AC-V (resp. SC-V) is better than AV-B (resp. SC-B) in terms of the running time while it is opposite in terms of the communication overhead. This is due to the effect of *straggler* [2]. Specifically, for the block-centric framework, in each iteration, the algorithms use block information to converge locally (i.e., within each block); they cannot start the next iteration until all blocks have converged. There are machines where some blocks may converge very slowly, which deteriorates the overall performance of the block-centric algorithms.

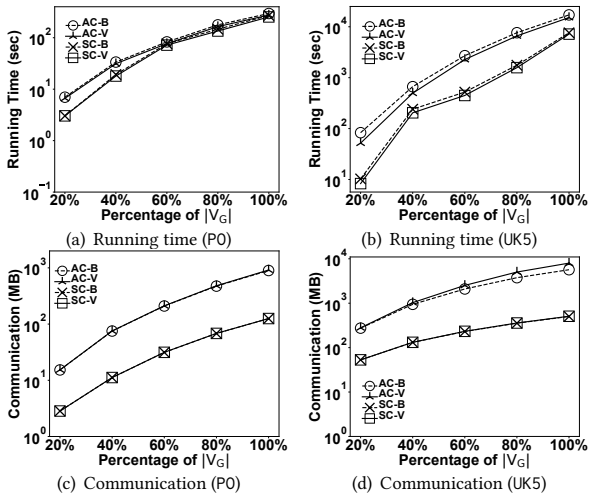
Exp-4: Effect of the number of machines. Next, we vary the number of machines from 2 to 16 and test its effect on performance. Figure 5 reports the results for datasets UK2 and HW. As shown in Figures 5(a) and 5(b), all of our algorithms take less running time when the number of machines increases. This is because the more the machines, the stronger the computing power our algorithms can take advantage of, thanks to their distributed designs. In addition, Figures 5(c) and 5(d) show that the communication overheads of all algorithms do not change with the number of machines. This is because the communication overhead is determined by the convergence rate of the algorithms, which is not influenced by the number of machines.

Exp-5: Effect of dataset cardinality. We evaluate the effect of cardinality for our proposed algorithms on datasets PO and UK5. For



(a) Running time (HW) (b) Running time (UK2)
(c) Communication (HW) (d) Communication (UK2)

Figure 5: Effect of # machines



(a) Running time (PO) (b) Running time (UK5)
(c) Communication (PO) (d) Communication (UK5)

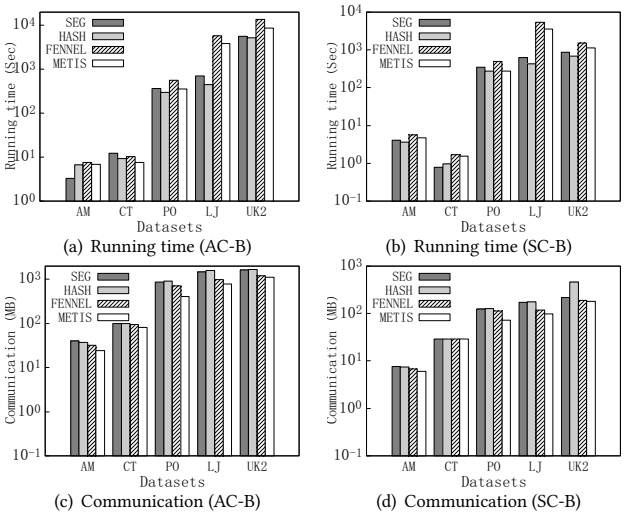
Figure 6: Effect of cardinality

this purpose, we extract a set of subgraphs from the original graphs by randomly selecting different fractions of vertices, which varies from 20% to 100%. As shown in Figure 6, both the running time and communication overhead increase with the dataset cardinality. This is expected because the larger the dataset, the more the corenesses of the vertices to compute, resulting in to poorer performance.

Exp-6: Effect of partition strategies. We evaluate the effect of different partition strategies in block-centric algorithms, i.e., AC-B and SC-B. Specifically, we compare four partition strategies, including SEG [12], HASH [12], FENNEL [42], and METIS [21].

- SEG is a built-in partitioner of GRAPE. Let C be the maximum cardinality of partitioned subgraphs. For a vertex v with its ID $v_{id} \in [0, n-1]$, v is allocated to the i-th subgraph, where $i = v_{id}/C$.
- HASH is also a built-in partitioner of GRAPE. Let N be the number of partitioned subgraphs. For a vertex v with its ID $v_{id} \in [0, n-1]$, v is allocated to the i-th subgraph, where $i = v_{id}\%N$.
- FENNEL subsumes two popular heuristics to partition the graph: the folklore heuristic that places a vertex to the subgraph with the fewest non-neighbors, and the degree-based heuristic that uses different heuristics to place a vertex based on its degree.
- METIS is a popular edge-cut partitioner that partitions the graph into subgraphs with minimum crossing edges.

Figure 7 shows the results. We can observe that HASH has the best performance in terms of running time on most datasets, but it has the worse performance in terms of communication cost. This is because HASH has more balanced partitions (i.e., each partition has almost an equal number of vertices) while METIS and FENNEL have higher locality, which leads to more running time, due to the effect of *straggler*, but less communication overhead. Considering the importance of efficiency in practice, we employ HASH as the default partition strategy in our experiments, as mentioned earlier.



(a) Running time (AC-B) (b) Running time (SC-B)
(c) Communication (AC-B) (d) Communication (SC-B)

Figure 7: Effect of partition strategies

7 CONCLUSION

In this paper, we study the problem of D-core decomposition in distributed settings. To handle distributed D-core decomposition, we propose two efficient algorithms, i.e., the anchored-coreness-based algorithm and skyline-coreness-based algorithm. Specifically, the anchored-coreness-based algorithm employs in-H-index and out-H-index to compute the anchored corenesses in a distributed way; the skyline-coreness-based algorithm uses a newly designed index, called D-index, for D-core decomposition through skyline coreness computing. Both theoretical analysis and empirical evaluation show the efficiency of our proposed algorithms with fast convergence rates.

As for future work, we are interested to study how to further improve the performance of the skyline-coreness-based algorithm, in particular how to accelerate the computation of D-index on each single machine. We are also interested to investigate efficient algorithms of distributed D-core maintenance for dynamic graphs.

ACKNOWLEDGMENTS

This work is supported by Hong Kong RGC Projects C2004-21GF, C6030-18GF, 12202221, 22200320, 12200021, 12201119, 12201518, and RIF Project R2002-20F. Jianliang Xu is the corresponding author.

REFERENCES

[1] 2012. Giraph. https://giraph.apache.org/.

[2] Ganesh Ananthanarayanan, Srikanth Kandula, Albert G. Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. 2010. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *OSDI*. 265–278.

[3] Sabeur Aridhi, Martin Brugnara, Alberto Montresor, and Yannis Velegrakis. 2016. Distributed k-core decomposition and maintenance in large dynamic graphs. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. 161–168.

[4] Vladimir Batagelj and Matjaz Zaversnik. 2003. An O (m) algorithm for cores decomposition of networks. *arXiv preprint cs/0310049* (2003).

[5] Francesco Bonchi, Francesco Gullo, Andreas Kaltenbrunner, and Yana Volkovich. 2014. Core decomposition of uncertain graphs. In *Proceedings of the 20th International Conference on Knowledge Discovery and Data Mining*. 1316–1325.

[6] Francesco Bonchi, Arijit Khan, and Lorenzo Severini. 2019. Distance-generalized core decomposition. In *Proceedings of the 2019 International Conference on Management of Data*. 1006–1023.

[7] Yankai Chen, Jie Zhang, Yixiang Fang, Xin Cao, and Irwin King. 2020. Efficient community search over large directed graphs: An augmented index-based approach. In *International Joint Conference on Artificial Intelligence*. 3544–3550.

[8] James Cheng, Yiping Ke, Shumo Chu, and M Tamer Özsu. 2011. Efficient core decomposition in massive networks. In *2011 IEEE 27th International Conference on Data Engineering*. 51–62.

[9] Naga Shailaja Dasari, Ranjan Desh, and Mohammad Zubair. 2014. ParK: An efficient algorithm for k-core decomposition on multicore processors. In *2014 IEEE International Conference on Big Data*. 9–16.

[10] Marius Eidsaa and Eivind Almaas. 2013. S-core network decomposition: A generalization of k-core analysis to weighted networks. *Physical Review E* 88, 6 (2013), 062819.

[11] Hossein Esfandiari, Silvio Lattanzi, and Vahab Mirrokni. 2018. Parallel and streaming algorithms for k-core decomposition. In *International Conference on Machine Learning*. 1397–1406.

[12] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, and Jiaxin Jiang. 2017. GRAPE: Parallelizing sequential graph computations. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1889–1892.

[13] Yixiang Fang, Zhongran Wang, Reynold Cheng, Hongzhi Wang, and Jiafeng Hu. 2018. Effective and efficient community search over large directed graphs. *IEEE Transactions on Knowledge and Data Engineering* 31, 11 (2018), 2093–2107.

[14] Yixiang Fang, Yixing Yang, Wenjie Zhang, Xuemin Lin, and Xin Cao. 2020. Effective and Efficient Community Search over Large Heterogeneous Information Networks. *Proceedings of the VLDB Endowment* 13, 6 (2020), 854–867.

[15] Edoardo Galimberti, Martino Ciaperoni, Alain Barrat, Francesco Bonchi, Ciro Cattuto, and Francesco Gullo. 2021. Span-core Decomposition for Temporal Networks: Algorithms and Applications. *ACM Transactions on Knowledge Discovery from Data* 15, 1 (2021), 2:1–2:44.

[16] David Garcia, Pavlin Mavrodiev, Daniele Casati, and Frank Schweitzer. 2017. Understanding popularity, reputation, and social influence in the twitter society. *Policy & Internet* 9, 3 (2017), 343–364.

[17] Christos Giatsidis, Dimitrios M Thilikos, and Michalis Vazirgiannis. 2013. D-cores: measuring collaboration of directed graphs based on degeneracy. *Knowledge and Information Systems* 35, 2 (2013), 311–343.

[18] Jorge E. Hirsch. 2005. H-index. https://en.wikipedia.org/wiki/H-index/.

[19] Xin Huang, Laks VS Lakshmanan, and Jianliang Xu. 2017. Community search over big graphs: Models, algorithms, and opportunities. In *Proceedings of the 33rd IEEE International Conference on Data Engineering*. 1451–1454.

[20] Xin Huang, Laks VS Lakshmanan, and Jianliang Xu. 2019. *Community Search over Big Graphs*. Morgan & Claypool Publishers.

[21] George Karypis and Vipin Kumar. 1998. Metis: A software package for partitioning unstructured graphs. *Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version* 4 (1998).

[22] Wissam Khaouid, Marina Barsky, Venkatesh Srinivasan, and Alex Thomo. 2015. K-core decomposition of large networks on a single pc. *Proceedings of the VLDB Endowment* 9, 1 (2015), 13–23.

[23] Xunkun Liao, Qing Liu, Jiaxin Jiang, Xin Huang, Jianliang Xu, and Byron Choi. 2022. Distributed D-core Decomposition over Large Directed Graphs. In *arXivpreprint arXiv:2202.05990*.

[24] Nathan Linial. 1992. Locality in Distributed Graph Algorithms. *SIAM J. Comput.* 21, 1 (1992), 193–201.

[25] Boge Liu, Long Yuan, Xuemin Lin, Lu Qin, Wenjie Zhang, and Jingren Zhou. 2019. Efficient (a,β)-core Computation: an Index-based Approach. In *International World Wide Web Conference*. 1130–1141.

[26] Qing Liu, Minjun Zhao, Xin Huang, Jianliang Xu, and Yunjun Gao. 2020. Truss-based community search over large directed graphs. In *Proceedings of the 2020 ACM International Conference on Management of Data*. 2183–2197.

[27] Qing Liu, Xuliang Zhu, Xin Huang, and Jianliang Xu. 2021. Local Algorithms for Distance-generalized Core Decomposition over Large Dynamic Graphs. *The Proceedings of the VLDB Endowment* (2021).

[28] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. 2012. Distributed graphlab: A framework for machine learning in the cloud. *arXiv preprint arXiv:1204.6078* (2012).

[29] Linyuan Lü, Tao Zhou, Qian-Ming Zhang, and H Eugene Stanley. 2016. The H-index of a network node and its relation to degree and coreness. *Nature Communications* 7, 1 (2016), 1–7.

[30] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 International Conference on Management of Data*. 135–146.

[31] Aritra Mandal and Mohammad Al Hasan. 2017. A distributed k-core decomposition algorithm on spark. In *IEEE International Conference on Big Data*. 976–981.

[32] Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing. *Comput. Surveys* 48, 2 (2015), 25:1–25:39.

[33] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. 2012. Distributed k-core decomposition. *IEEE Transactions on Parallel and Distributed Systems* 24, 2 (2012), 288–300.

[34] You Peng, Ying Zhang, Wenjie Zhang, Xuemin Lin, and Lu Qin. 2018. Efficient Probabilistic K-Core Computation on Uncertain Graphs. In *IEEE International Conference on Data Engineering*. 1192–1203.

[35] Semih Salihoglu and Jennifer Widom. 2013. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*. 1–12.

[36] Ahmet Erdem Saríyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V Çatalyürek. 2013. Streaming algorithms for k-core decomposition. *Proceedings of the VLDB Endowment* 6, 6 (2013), 433–444.

[37] Ahmet Erdem Sarıyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V Çatalyürek. 2016. Incremental k-core decomposition: algorithms and evaluation. *The VLDB Journal* 25, 3 (2016), 425–447.

[38] Ahmet Erdem Sariyüce, C. Seshadhri, and Ali Pinar. 2018. Local Algorithms for Hierarchical Dense Subgraph Discovery. *Proceedings of the VLDB Endowment* 12, 1 (2018), 43–56.

[39] Stephen B Seidman. 1983. Network structure and minimum degree. *Social Networks* 5, 3 (1983), 269–287.

[40] Henry Soldano, Guillaume Santini, Dominique Bouthinon, and Emmanuel Lazega. 2017. Hub-authority cores and attributed directed network mining. In *International Conference on Tools with Artificial Intelligence*. 1120–1127.

[41] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From" think like a vertex" to" think like a graph". *Proceedings of the VLDB Endowment* 7, 3 (2013), 193–204.

[42] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. 2014. FENNEL: Streaming Graph Partitioning for Massive Scale Graphs. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*. 333–342.

[43] Dong Wen, Lu Qin, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2016. I/O efficient Core Graph Decomposition at web scale. In *International Conference on Data Engineering*. 133–144.

[44] Huanhuan Wu, James Cheng, Yi Lu, Yiping Ke, Yuzhen Huang, Da Yan, and Hejun Wu. 2015. Core decomposition in large temporal graphs. In *IEEE International Conference on Big Data*. 649–658.

[45] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2014. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1981–1992.

[46] Wei Zhou, Hong Huang, Qiang-Sheng Hua, Dongxiao Yu, Hai Jin, and Xiaoming Fu. 2021. Core decomposition and maintenance in weighted graph. *World Wide Web* 24, 2 (2021), 541–561.