Efficient Parallel D-core Decomposition at Scale

Wensheng Luo
The Chinese University of Hong Kong, Shenzhen
luowensheng@cuhk.edu.cn

Yixiang Fang
The Chinese University of Hong Kong, Shenzhen
fangyixiang@cuhk.edu.cn

Chunxu Lin
The Chinese University of Hong Kong, Shenzhen
chunxulin1@link.cuhk.edu.cn

Yingli Zhou
The Chinese University of Hong Kong, Shenzhen
yinglizhou@link.cuhk.edu.cn

ABSTRACT

Directed graphs are prevalent in social networks, web networks, and communication networks. A well-known concept of the directed graph is the D-core, or (k, l)-core, which is the maximal subgraph in which each vertex has an in-degree not less than k and an out-degree not less than l. Computing the non-empty D-cores for all possible values of k and l, a.k.a. D-core decomposition, has found versatile applications spanning social network analysis, community search, and graph visualization. However, existing algorithms of D-core decomposition suffer from efficiency and scalability issues on large graphs, because serial peeling-based algorithms are limited by single-core utilization, while skyline coreness-based methods exhibit notably high time complexity. To tackle these issues, in this paper we propose efficient parallel algorithms for D-core decomposition by leveraging the computational prowess of multicore CPUs. Specifically, we first propose a novel algorithm that computes the D-cores for each possible k value, by exploiting an implicit level-by-level vertex removal strategy, which not only diminishes dependencies between vertices but also maintains a time complexity akin to that of sequential algorithms. We further develop an advanced algorithm by introducing a novel concept of D-shell, which allows us to curtail redundant computations by reducing the necessary k values when computing corresponding D-cores, and deriving D-cores with larger k values from the D-cores currently computed based on D-shell. Extensive experiments on ten real-world large graphs show that our algorithms are highly efficient and scalable, and the advanced algorithm is up to two orders of magnitude faster than the state-of-the-art parallel decomposition algorithm with 32 threads.

PVLDB Reference Format:
Wensheng Luo, Yixiang Fang, Chunxu Lin, and Yingli Zhou. Efficient Parallel D-core Decomposition at Scale. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/GearlessL/PDC.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

1 INTRODUCTION

As a fundamental data structure, directed graphs are able to capture the complex relationships between entities through directed edges. Directed graphs are prevalent in social networks, web networks, and communication networks. For example, in social networks (e.g., Facebook, Twitter, and Instagram), the directed graph can showcase users' following-follower relationships; in the World Wide Web, it maps hyperlink connections between web pages as directed edges; in communication networks, it is able to model the information transmission between nodes. A well-known concept of the directed graph is the D-core [21], or (k, l)-core, which is the maximal subgraph in which each vertex has an in-degree not less than k and an out-degree not less than l. This concept was extended from the classic k-core on the undirected graph, where the k-core is defined as the maximal subgraph in which each vertex has a degree of k or more [3, 39, 43]. For instance, in the directed graph depicted in Figure 1, the subgraph comprising $\{v_3, v_5, v_6, v_8\}$ constitutes a $(3, 3)$-core, because every vertex has both an in-degree and an out-degree of 3 or more. Conversely, the subgraph formed by the subset $\{v_2, v_3, v_5, v_6, v_7, v_8\}$ creates a $(3, 1)$-core, where each vertex maintains an in-degree of at least 3. However, vertex v_2 in this subgraph exhibits an out-degree of 1.

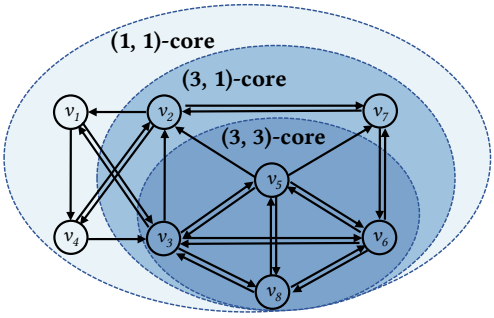


Figure 1: Illustrating D-cores on a directed graph.

Given a directed graph, computing the non-empty D-cores for all possible values of k and l is referred to as D-core decomposition. D-core decomposition has found extensive applications, such as community search [8, 16], graph visualization [1], social network influence assessment [19], network centrality and authority identification [45], evaluation of graph collaboration features [21], to name a few. In the following, we present three concrete applications:

- ***Community search.*** Given a query vertex q of a directed graph, community search aims to find the most-likely community containing q, which has found many applications such as friend recommendation, event organization, and biological network analysis. As shown in [8, 16], D-core is effective for modelling communities in the directed graph. The in-degree and out-degree constraints of D-core allows us to identify the highly interactive communities, ensuring that each vertex within the returned community showcases significant interactions and robust cohesion with other members, aligning with the expectations of the query users.
- ***Networks visualization.*** For undirected graphs, the k-core decomposition unveils the hierarchical and topological attributes of vertices within a graph, making it a valuable tool for visualizing undirected graphs [1]. As the directed version of k-core, D-core with higher k and l values distinctly correlate with vertices holding elevated degrees and more central positions within the network structure. This functionality enables visualization tools to discern genuine or computer-generated network patterns based on attributes like hierarchical layout, degree correlations, and centrality.
- ***Network influence evaluation.*** As shown in [19], Garcia et al. employed extensive digital data from 2009 to 2016 to explore the realms of popularity, reputation, and social influence on Twitter. By analyzing network information from over 40 million users, they derived new global reputation measures by leveraging D-core decomposition and the bow-tie structure of the Twitter follower network. By computing D-cores, they quantify reputation on a global scale, capturing the recursive nature of reputation as a centrality measure: Users in (k, l)-cores with high k are followed by users also in (k, l)-cores with high k, illustrating the propagation of influence.

To compute all D-cores, Giatsidis et al. [21] introduced a peeling-based algorithm that iteratively eliminates vertices with in-degrees less than k and out-degrees less than l. However, it is very inefficient for large graphs, since it requires computing the D-core for all possible values of k and l across the entire graph. To enhance efficiency, Fang et al. [16] introduced an improved D-core decomposition algorithm. The approach first calculates the maximum value of k, denoted as k_{max}, in the graph, and then for each k within the range $[0, k_{max}]$, it computes all D-cores associated with the given k by removing vertices with the smallest out-degrees and in-degrees less than k. Utilizing optimization techniques, such as binsort derived from k-core decomposition [3], the time complexity for computing all D-cores corresponding to k is $O(m)$, where m is the number of edges in the graph. Considering the upper bound for k_{max} as $O(\sqrt{m})$, the overall time complexity stands at $O(m\sqrt{m})$.

While peeling-based algorithms demonstrate favorable time complexity, their efficiency declines as graph sizes increase, due to their serial nature, which constrains their capacity to leverage the computational resources of multiple CPU cores. The main challenge stems from the intrinsic non-parallelizability of peeling-based algorithms, because the updates of out-degree and in-degree for vertices rely on previously deleted vertices throughout the entire process, thereby establishing a significant dependency that permeates the entirety of the procedure.

To enhance the parallel efficiency of D-core decomposition, Liao et al. [30] proposed a distributed parallel algorithm for D-core decomposition, which leverages the local information of each vertex to compute D-core. Specifically, the authors introduced the concept of coreness for directed graph vertices, referred to as skyline coreness. In contrast to coreness in undirected graphs, which signifies the maximum value of k corresponding to a non-empty k-core containing the vertex, skyline coreness comprises a set of integer pairs, which record the values of k and l associated with D-cores encompassing the vertex and not subsumed by other D-cores. Similar to h-index-based algorithms for k-core [41], skyline coreness is updated iteratively through neighbor information for each vertex until convergence, effectively avoiding information synchronization between vertices and thereby improving algorithm parallelism. However, the computation and update of skyline coreness are time-consuming, leading to time complexity of $O(d_{\max} \cdot d^{+}_{\max} \cdot m)$, where $d_{\max}$ and $d^{+}_{\max}$ are the maximum degree and maximum out-degree among all vertices, respectively. Notice that this complexity is significantly higher than that of peeling-based algorithms, posing challenges in handling large-scale graphs.

Our technical contributions. To improve the efficiency of D-core decomposition, in this paper, we aim to develop efficient parallel algorithms under the shared-memory model. Inspired by a parallel k-core decomposition algorithm [26], we propose a novel parallel D-core decomposition algorithm, called `ParPeel`. Specifically, `ParPeel` first parallelly computes the maximum value of k, denoted by k_{max}, of the graph, and then for each k in $[0, k_{max}]$, it computes all (k, l)-cores where l ranges from 0 to its maximum value. To reduce dependencies between vertices during the computation, we employ an implicit vertex removal strategy in a level-by-level manner. For a given k, we first initiate the level at 0, and then scan the graph for vertices with an out-degree equal to the current level and parallelly remove them. Subsequently, we update the degrees of all removed vertices' neighbors and rescan the remaining vertices to eliminate those with an out-degree equal to the current level and an in-degree less than k. This process continues until there are no such vertices in the graph. Afterwards, we increase the level value and repeat the above steps until all vertices have been processed. Interestingly, `ParPeel` not only reduces dependencies between vertices but also maintains a time complexity close to that of sequential algorithms.

Despite the superior performance of `ParPeel` compared to existing algorithms, it faces challenges related to redundant computations. This arises from the fact that a vertex can belong to multiple (k, l)-cores, necessitating repetitive scanning of vertices in the graph by `ParPeel`. Moreover, there is a potential limitation in fully utilizing the multi-core computing power when the scale of the graph being processed gradually decreases, as the value of k increases. To further enhance efficiency, we propose an advanced algorithm, called `Shell-PDC`. Specifically, we first introduce a novel hierarchical structure of D-core, termed as (k, l)-shell or D-shell. Given a specific l, the (k', l)-shell is the set of vertices exclusively existing in the (k', l)-core and is absent in other (k, l)-cores corresponding to k values greater or smaller than the current k'. Subsequently, we show that instead of computing D-cores for k_{max} different k values, we actually only need to calculate the D-cores corresponding to the k values associated with each distinct

(k,l)-shell. As a result, the redundant computation can be reduced dramatically, particularly given that the number of (k,l)-shells is smaller than k_{max} in real-world datasets, especially in skewed graphs. Furthermore, we explore the relationships between (k,l)-cores with different k values, and propose an iterative algorithm to compute larger (k,l)-cores based on the smaller ones obtained for the current k, eliminating the need for redundant graph peeling operations. This not only reduces computational redundancy to improve algorithm efficiency but also enhances thread utilization, leading to better parallel performance.

We have implemented all our algorithms by OpenMP[1], a popular shared-memory parallel processing framework. We have also conducted extensive experiments using ten real-world large directed graphs, five of which consist of more than 100 million edges. Experimental results show that our advanced parallel algorithm `Shell-PDC` outperforms state-of-the-art parallel D-core decomposition algorithms, achieving a remarkable speedup of up to two orders of magnitude with 32 threads.

Outline. We review related works in Section 2, formally introduce the D-core decomposition problem in Section 3, and present our proposed `ParPeel` algorithm in Section 4. Our advanced decomposition algorithm `Shell-PDC` is detailed in Section 5. We report the experimental results in Section 6, and conclude in Section 7.

2 RELATED WORK

In this section, we mainly review the existing works of decomposition for cores and other cohesive subgraphs in large graphs.

2.1 Core decomposition

k-core is one of the most representative cohesive subgraph models [6, 33]. For an undirected graph and a given threshold k, the k-core is a maximal subgraph where each vertex has a degree of at least k. Finding all k-cores corresponding to all possible values of k in a graph is known as core decomposition. Currently, there are many efficient algorithms for core decomposition. Batagelj et al. [3] proposed a core decomposition algorithm based on binsort which has a time complexity linear to the number of edges in the graph. Kabir et al. [26] introduced a shared-memory parallel core decomposition algorithm that improves efficiency by batch-deleting vertices with the smallest degrees from the graph. Sariyuce et al. [41] proposed a parallel core decomposition algorithm based on the h-index [33], where each vertex updates its h-index using neighbor information until convergence. Dhulipala et al. [X] introduced a framework centered around work-efficient bucketing for parallel algorithms, deploying it to implement the parallel k-core decomposition algorithm. Following this, Huang et al. [X] expanded upon the framework to tackle the bi-core decomposition problem in bipartite graphs. However, these methodologies are not directly applicable to solving D-core decomposition due to the distinct structural characteristics of D-core in contrast to k-core or bi-core. Additionally, there are distributed [2, 39], streaming [13, 40], and disk-based [9, 27] core decomposition algorithms.

In directed graphs, as the degree of a vertex is divided into out-degree and in-degree, the D-core model is proposed [21]. Fang et al. [16] presented a single-machine D-core decomposition algorithm with a time complexity of $O(m\sqrt{m})$. Liao et al. [30] proposed a distributed parallel algorithm that calculates the D-core for each vertex based on the locally computed skyline coreness, which enhances parallelism and reduces communication overhead. However, the time complexity of this method is increased, and its performance on large graphs needs further improvement.

Furthermore, core models and their decompositions on bipartite graphs [31, 36], uncertain graphs [4], temporal graphs [51], and heterogeneous information networks [15, 17] have also been extensively researched.

2.2 Decomposition of other cohesive subgraphs

In addition to k-core, there are several other typical cohesive subgraphs in undirected graphs, such as k-truss [22, 48], k-clique [11, 28], nucleus [42, 44], k-edge connected components [7, 20], and densest subgraph [18, 35]. Some subgraph models have also been extended to other types of graphs. For instance, D-truss [32] and densest subgraph [38] in directed graphs, bi-truss [49] and biclique [37] in bipartite graphs, (k,γ)-truss [23, 46], (k,τ)-clique [10, 29] in uncertain graphs. However, due to the differences in these models, their methods cannot be directly applied to D-core decomposition.

There are also several other typical cohesive subgraphs exist in undirected graphs. For instance, the k-truss [22] requires that each edge in the subgraph is contained in at least $k-2$ triangles. The k-clique [24, 28] demands that k vertices in the subgraph are fully connected, and the densest subgraph [14, 18] is defined as the subgraph with the largest density among all subgraphs, where density is the ratio of the number of edges to the number of vertices. These subgraph models have also found extensions to directed graphs. For example, D-truss [32, 47] is the directed version of k-truss with two types of triangles, and directed densest subgraph [35, 38] extends undirected density to directed density. Furthermore, cohesive subgraph models in other types of graphs have been studied. For example, bi-truss [49], biclique [37, 50], and quasi-biclique [34] in bipartite graphs, as well as (k,γ)-truss [46], (k,τ)-clique [10] in uncertain graphs. However, due to the differences in these models, their methods cannot be directly applied to D-core decomposition.

3 PROBLEM STATEMENT

Let $G=(V,E)$ be a directed, unweighted simple graph, where V and E are the sets of vertices and edges of G, respectively. Any edge $e=(u,v)$ in E is directed, meaning that e is an edge from u to v, and we call u the in-neighbor of v, and v the out-neighbor of u. Correspondingly, we denote the sets of out-neighbors and in-neighbors of vertex v in V as $N^-(v)$ and $N^+(v)$, respectively. The number of out-neighbors and in-neighbors of vertex v is called its out-degree and in-degree, respectively, and is denoted as $d^-(v)$ and $d^+(v)$, i.e., $d^-(v)=|N^-(v)|$, $d^+(v)=|N^+(v)|$. Additionally, we define the degree of v as the sum of its out-degree and in-degree, that is, $d(v)=d^-(v)+d^+(v)$. Accordingly, we denote the maximum value of out-degree in G as d^-_{max} , the maximum value of in-degree as d^+_{max}, and the maximum value of degree as d_{max}. Table 1 shows the notations frequently used and their meanings in this paper.

Based on the in-degree and out-degree of vertices, the definition of D-core is as follows.

[1] https://www.openmp.org/

Table 1: Notations and meanings.

Notation	Meaning
$G = (V, E)$	A directed graph G with vertex set V and edge set E
n, m	The numbers of vertices and edges in G resp.
$N^-(v)$, $N^+(v)$	The out-neighbor set and in-neighbor set of a vertex $v \in G$ resp.
$d(v)$	The degree of a vertex $v \in G$
d_{max}	The maximum degree of all vertices in G
$d^-(v)$, $d^+(v)$	The out-degree and in-degree of a vertex $v \in G$ resp.
d^-_{max}, d^+_{max}	The maximum out-degree and in-degree of all vertex in G resp.
k_{max}, l_{max}	The maximum k and l among all D-cores of G resp.
$K(v, l)$, $L(v, k)$	The in-core number and out-core number of a vertex $v \in G$ for a given l and k resp.

Definition 1 (D-core [21]). *Given a directed graph $G = (V, E)$ and two integers k and l, a D-core of G, also known as a (k, l)-core, is a maximal subgraph $H = (V_H, E_H) \subseteq G$ such that the in-degree and out-degree of vertices in H are not less than k and l, respectively, i.e., $\forall v \in V_H, d^+(v) \geq k$ and $d^-(v) \geq l$.*

We refer to (k, l) as the *d-pair* of the D-core, where k and l are respectively called the in-core number and out-core number of the (k, l)-core. For all D-cores, we record the maximum value that k can obtain as k_{max}, and the maximum value of l is l_{max}. For a vertex v in G, when the constraints of the given in-degree/out-degree are different, the out-core/in-core number is also different. We denote the out-core/in-core number of v for a given k/l as $L(v, k)/K(v, l)$.

Similar to the classic k-core [3, 43], the D-core also has some interesting properties: (1) a D-core is not necessarily connected; (2) for any d-pair (k, l), there is at most one (k, l)-core in G; and (3) the D-cores have the partially nested relationship which is stated as follows.

Property 1. *Given a directed graph G and two D-cores, say (k_1, l_1)-core and (k_2, l_2)-core, if $k_1 \geq k_2$ and $l_1 \geq l_2$, then the (k_1, l_1)-core is a subgraph of the (k_2, l_2)-core.*

Problem statement. In this paper, we study the D-core decomposition problem. Specifically, given a directed graph G, D-core decomposition aims to find all D-cores in G, that is, to find the non-empty (k, l)-cores corresponding to all possible d-pairs.

Table 2: All non-empty D-cores of the directed graph shown in Figure 1.

l \ k	0	1	2	3
0	$v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8$	$v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8$	$v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8$	$v_2, v_3, v_5, v_6, v_7, v_8$
1	$v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8$	$v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8$	$v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8$	$v_2, v_3, v_5, v_6, v_7, v_8$
2	$v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8$	$v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8$	$v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8$	v_3, v_5, v_6, v_8
3	v_3, v_5, v_6, v_8	v_3, v_5, v_6, v_8	v_3, v_5, v_6, v_8	v_3, v_5, v_6, v_8

Example 1. *Figure 1 shows a directed graph G with 8 vertices, and Table 2 illustrates all the D-cores of G. The values of k_{max} and l_{max} are both 3, resulting in a total of 9 distinct D-cores with different k and l values. Each cell in the table represents the vertices in the corresponding D-core. For example, a subgraph induced by $\{v_3, v_5, v_6, v_8\}$ represents a (3, 3)-core of G, and the subgraph induced by $\{v_2, v_3, v_5, v_6, v_7, v_8\}$ is a (3, 1)-core of G. According to Property 1, we can observe that (3,3)-core ⊆ (3,2)-core ⊆ (3,1)-core ⊆ (3,0)-core. Likewise, (3,3)-core ⊆ (2,3)-core ⊆ (1,3)-core ⊆ (0,3)-core. The D-core decomposition aims to obtain all the D-cores of G, covering all possible pairs of k and l values.*

Computation model. In this paper, we employ the work-span model to analyze the algorithms, which is a widely utilized framework in the analysis of shared-memory algorithms [25]. The operations of a parallel algorithm form a directed acyclic graph, where each vertex represents an operation and directed edges denote dependencies between operations (see Figure 2). The *work* of an algorithm corresponds to its total number of operations, reflecting its time complexity. Conversely, the *span* (or depth) represents the longest dependency path in the algorithm. For instance, in Figure 2, the algorithm's work is 10, with a span of 4. The theoretical running time of a parallel algorithm can be expressed as $work/p + span$ [5], where p denotes the number of threads utilized.

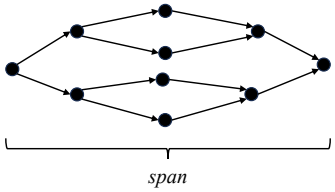


Figure 2: An example of the work-span computation model.

4 A PARALLEL D-CORE DECOMPOSITION ALGORITHM

In this section, we begin by introducing the state-of-the-art D-core decomposition algorithms and delving into its limitations. Following this, we present a novel and efficient parallel D-core decomposition algorithm.

4.1 State-of-the-art algorithms

The state-of-the-art serial D-core decomposition algorithm, named `Peeling`, is introduced by [16]. The algorithm involves computing (k,l)-cores for each k, with k ranging from 0 to the maximum in-degree of the graph. Specifically, for each k, the procedure operates sequentially, removing disqualified vertices from the graph through an iterative process that eliminates those with the smallest out-degrees. Despite its efficiency in obtaining D-cores within $O(m\sqrt{m})$ time using techniques like binsort [3], its parallelization faces challenges. The iterative process of identifying and removing vertices with the smallest out-degrees, along with updating both out-degrees and in-degrees of the remaining vertices until the graph is empty, makes it inherently non-parallelizable due to the dependencies in the updating of degrees among vertices. Therefore, while `Peeling` is work-efficient, there remains a need for an efficient parallel D-core decomposition algorithm.

The state-of-the-art parallel approach leverages the partial nested property of D-core to introduce the concept of skyline coreness [30].

Definition 2 (skyline coreness [30]). *Given a directed graph $G(V, E)$, the skyline coreness of a vertex $v \in V$ is represented by an integer pair (k, l). This pair indicates that v is part of a maximal (k, l)-core, and there is no other (k', l')-core that contains it, where both k' and l' are greater than k and l, i.e., (k, l) is dominated by (k', l').*

Given a directed graph, each vertex in the graph has one or more skyline corenesses. For a vertex's skyline coreness (k, l), it must satisfy that it has at least k in-neighbors with a skyline coreness that dominates or is equal to (k, l), and at least l out-neighbors with a skyline coreness that dominates or is equal to (k, l).

Algorithm 1: SC [30]

Input: A directed graph $G = (V, E)$
Output: The skyline corenesses of all the vertices of G

```
1  Compute k_max and l_max for all vertices;
2  foreach v ∈ V do D_v ← {(k_max(v), l_max(v))};
3  F ← true;
4  while F do
5      F ← false;
6      for v ∈ V in parallel do
7          D ← ∅, k_max, l_max ← maximum k and l in D_v resp.;
8          for k ← k_max to 1 do
9              l ← l_max, l_min ← 0;
10             while l > l_min do
11                 if (k, l) meets the skyline coreness constraints then
12                     D ← D ∪ {(k, l)};
13                     l_min ← l;
14                 l ← l − 1;
15         if D_v ≠ D then F ← true, D_v ← D ;
16 return {D_v | v ∈ V} ;
```

Based on skyline coreness, Liao et al. [30] proposed an iterative algorithm that relies on vertex neighborhood information. The pseudo code for the algorithm is presented as Algorithm 1. Specifically, the algorithm starts by computing the maximum values of k and l for each vertex v, denoted as $k_{max}(v)$ and $l_{max}(v)$ respectively (line 1). Then, the skyline coreness of v is initialized as $(k_{max}(v), l_{max}(v))$, where D_v is the skyline coreness set of v (line 2). For each vertex, the algorithm retrieves the maximum values of k and l from its set of skyline corenesses. It then iterates over all d-pairs and examines whether the vertex satisfies the constraints of skyline coreness based on its neighbor information (lines 7-14). The algorithm updates the original skyline coreness with the obtained skyline coreness (line 15). This step is repeated until the coreness values of all vertices no longer change (line 4).

Example 2. *Taking the directed graph G in Figure 1 as an example, Table 3 illustrates the computation process of skyline coreness for each vertex. Here, $D^i(v)$ represents the skyline coreness value of vertex v in the i-th iteration. The initial skyline coreness for each vertex is determined by finding the maximum values of k and l that allow the vertex to be included in a $(k, 0)$-core and $(0, l)$-core, respectively. For instance, the initial skyline coreness of vertex v_2 is $(3, 2)$, indicating its inclusion in a (3, 0)-core and a (0, 2)-core. In each iteration, every vertex updates its skyline coreness value according to the constraints of skyline coreness until convergence. After two iterations, the skyline coreness of vertex v_2 converges to $\{(3, 1), (2, 2)\}$.*

Table 3: The computation process of skyline coreness for each vertex in Figure 1.

	v_1	v_2	v_3	v_4
$D^0(v)$	$\{(2,2)\}$	$\{(3,2)\}$	$\{(3,3)\}$	$\{(2,2)\}$
$D^1(v)$	$\{(2,2)\}$	$\{(3,1),(2,2)\}$	$\{(3,3)\}$	$\{(2,2)\}$
$D^2(v)$	$\{(2,2)\}$	$\{(3,1),(2,2)\}$	$\{(3,3)\}$	$\{(2,2)\}$
	v_5	v_6	v_7	v_8
$D^0(v)$	$\{(3,3)\}$	$\{(3,3)\}$	$\{(3,2)\}$	$\{(3,3)\}$
$D^1(v)$	$\{(3,3)\}$	$\{(3,3)\}$	$\{(3,1),(2,2)\}$	$\{(3,3)\}$
$D^2(v)$	$\{(3,3)\}$	$\{(3,3)\}$	$\{(3,1),(2,2)\}$	$\{(3,3)\}$

Limitations: Algorithm 1 reduces the interdependence among vertex computations by leveraging local information, and the order of updating skyline corenesses for each vertex does not affect the correctness of the final results. This property enhances parallelism. However, the algorithm still has limitations. Despite reducing computation dependencies, the work of Algorithm 1 increases. Specifically, the time complexity is given by $O(RSC \cdot d^+_{max} \cdot m)$, where RSC is the number of iterations in the algorithm and d^+_{max} represents the maximum out-degree among all vertices in the graph G. In comparison, the current state-of-the-art single-machine algorithm has a time complexity of $O(k_{max} \cdot m)$ [16], where k_{max} corresponds to d^+_{max} in the worst case. As a result, the work of Algorithm 1 grows exponentially, making it challenging to handle large-scale graphs, especially when k_{max} is large and the number of skyline corenesses per vertex is substantial.

4.2 Parallel D-core computation

To tackle the aforementioned challenges, we propose an efficient parallel D-core decomposition algorithm in this subsection. Before presenting the algorithm, we introduce the concept of a k-list as follows.

Definition 3 (k-list [16]). *Given a directed graph $G(V, E)$ and an integer k, a k-list is a list of all D-cores in G, where the in-core number of each D-core in the k-list is equal to k.*

Example 3. *Consider the directed graph in Figure 1, the D-core of each column in Table 2 generates the corresponding k-list for various values of k. For instance, when $k = 1$, the (1, 0)-core, (1, 1)-core, (1, 2)-core, and (1, 3)-core collectively constitute the 1-list. Besides, due to the property of partially nesting, decomposing the graph with a specific k value is adequate to derive the out-core number for all vertices, leading to the determination of the corresponding k-list.*

According to Definition 3, it is evident that all D-cores of G are formed by the k-lists for k in the range $[0, k_{max}]$. Consequently, decomposing D-cores necessitates the computation of all k-lists rather than individually determining each (k, l)-core. Fang et al. [16] devised a sequential algorithm based on global peeling to compute all k-lists. Specifically, it calculates the in-core number of each vertex for a given k by iteratively removing the vertex with the minimum out-degree. Our algorithm adheres to the same computational approach, as depicted in the workflow of our parallel D-core decomposition algorithm in Figure 3. Notably, the parallel computation

of the k-list is a subroutine of the entire D-core decomposition process. However, parallelizing the k-list computation presents inherent challenges due to sequential dependencies and the necessity for synchronization steps. To tackle this problem, drawing inspiration from parallel k-core algorithms [12, 26], we introduce a parallel k-list algorithm.

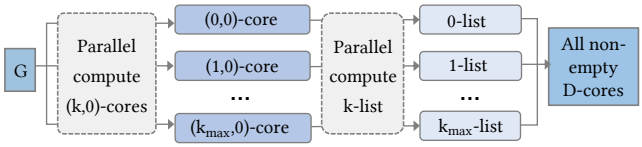


Figure 3: The workflow of our parallel D-core decomposition algorithm.

4.2.1 Parallel k-list computation. The main idea of the method is to peel or implicitly remove vertices with equal out-core numbers in parallel. Specifically, the method employs a level-by-level vertex processing strategy, where each level addresses vertices with the minimum out-degree and vertices with in-degrees less than k. Then the algorithm updates the out-degrees and in-degrees of the remaining vertices. This reduces the computation dependencies between vertices, thereby enhancing parallelism. Algorithm 2 provides the pseudo code for the algorithm.

We use the term *level* to represent the current out-core number, initialized as 0. Each thread has an array, buf, sized n/t (where n is total vertices and t is the number of threads) for storing vertices that need processing. *start* and *end* track the positions of vertices in buf. *visited*, initialized at 0, tracks the number of processed vertices (line 1), while $flag$, a boolean array set to *true* for all vertices in V, indicates unprocessed ones (line 2).

For all vertices in the graph, we first scan for those with out-degree equal to the current *level* or in-degree less than k (lines 4-9). These vertices are added to buf, and their flag is set to $false$. If a vertex has in-degree less than k, its out-degree is also set to *level* (line 9). For each vertex v in buf, we traverse its out-neighbors and in-neighbors (lines 11-23). For v's out-neighbor u, if u is unvisited, we decrement its in-degree. If the in-degree falls below k, we assign its out-core number as *level* and add it to buf (lines 12-16). For v's in-neighbor w, if its out-degree is greater than *level*, we decrement it (lines 18-19). If the decreased out-degree equals *level*, we add it to buf (lines 20-21). Note that vertices with out-degree less than *level* still have an out-core number of *level*, so we increment their out-degree by 1 to avoid errors (lines 22-23). It is important to note that despite each thread processing its own set of vertices within buf, there exists a possibility of multiple threads concurrently processing the same vertices when dealing with the neighbors of vertices in buf. Consequently, when modifying the out-degree and in-degree values of these vertices, it is imperative to employ atomic operations to ensure the integrity of the results and prevent any erroneous outcomes from arising. After processing all vertices at the current level, we count the processed vertices and increase *level* (lines 24-25). The algorithm terminates when the count equals n (line 3). The out-degree of all vertices represents the out-core number corresponding to the given k, forming the k-list.

Algorithm 2: `PKlist(k)`

Input: A directed graph $G = (V, E)$ and an integer k
Output: The k-list of G

```
1  level, start, end, visited ← 0, buf ← ∅;
2  foreach v ∈ V do flag[v] ← true;
3  while visited < n do
4      for v ∈ V ∧ flag[v] in parallel do
5          if d⁻(v) = level then
6              buf[end] ← v, end++, flag[v] ← false;
7          else if d⁺(v) < k then
8              buf[end] ← v, end++, flag[v] ← false;
9              d⁺(v) ← level;
10     while start < end do
11         v ← buf[start], start++, flag[v] ← false;
12         for u ∈ N⁻(v) ∧ flag[u] do
13             d ← atomicSub(d⁺(u),1);
14             if d ≤ k then
15                 buf[end] ← u, end++, flag[u] ← false;
16                 d⁻(u) ← level;
17         for w ∈ N⁺(v) ∧ flag[w] do
18             if d⁻(w) > level then
19                 d ← atomicSub(d⁻(w),1);
20                 if d = level + 1 then
21                     buf[end] ← w, end++, flag[w] ← false;
22                 if d ≤ level then
23                     atomicAdd(d⁻(w),1), flag[w] ← false;
24     atomicAdd(visited, end);
25     start, end ← 0, level ← level + 1;
26 return {d⁻(v) | v ∈ V} ;
```

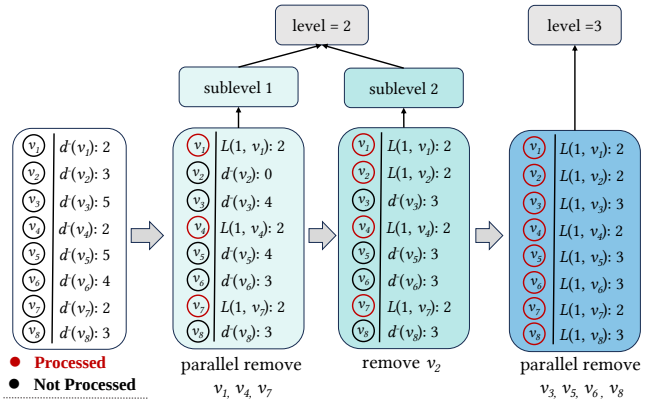


Figure 4: The process of parallel 1-list computation for the graph in Figure 1.

Example 4. *Consider the directed graph G as illustrated in Figure 1. Let us perform the computation of the 1-list with a parameter $k = 1$. The whole process is shown in Figure 4. It is important to note that all vertices in the graph have an in-degree of at least 1. The process commences by systematically scanning all vertices having an out-degree less than or equal to the current level, with the initialization*

of level = 0. In case there are no vertices satisfying this criterion, the level value is incremented by 1.

At level = 2, the initial scan involves vertices v_1, v_4, and v_7 since these three vertices exhibit an out-degree of 2. Subsequently, these vertices are removed from the graph, and adjustments are made to the in-degree and out-degree of their respective neighbors. The subsequent scan involves vertex v_2, which has an out-degree of less than 2. As for the remaining vertices, each of them possesses an out-degree greater than 2. Consequently, the level is increased by 1, and the scan proceeds to cover all vertices with an out-degree less than 3. Ultimately, we deduce that vertices v_3, v_5, v_6, and v_8 possess an out-core number of 3. This enables the algorithm to process multiple vertices concurrently without the necessity of individual vertex removal and waiting for the update of neighboring degrees.

Analysis. For the sequential algorithm [16], the time complexity of computing a k-list for a given k is $O(m)$, which can be accomplished using the binsort. In `PKlist`, a total of l_{max} scans are performed on the vertices of the graph, with each scan requiring $O(n)$ operations. Thus, the time complexity of the scan process is $O(l_{max} \cdot n)$. Additionally, the time complexity for updating the degrees of all vertices is $O(m)$. Consequently, the overall work of Algorithm 2 can be expressed as $O(l_{max} \cdot n + m)$ in the worst-case scenario.

4.2.2 Overall algorithm. Based on Algorithm 2, we can obtain the parallel D-core decomposition algorithm. The pseudocode of the parallel peeling-based D-core decomposition algorithm (`ParPeel`) is shown in Algorithm 3.

Algorithm 3: `ParPeel`

Input: A directed graph $G = (V, E)$
Output: All the k-lists of G

```
1 Compute k_max of G, Res ← ∅;
2 for k ∈ [0, k_max] do
3     Res ← Res ∪ PKlist(k) ;          // Algorithm 2
4 return Res ;
```

Specifically, we begin by determining the maximum value of k, denoted as $k_{\max}$, for all D-cores in the graph G. This computation is performed by setting l to 0 and evaluating all $(k, 0)$-cores in G. Since l is 0, the focus is solely on the out-neighbors and in-degrees of each vertex. Algorithm 2 can be applied to accomplish this, by swapping $d^+(v)$ and $N^+(v)$ with $d^-(v)$ and $N^-(v)$, respectively, and vice versa (line 1). Subsequently, Algorithm 2 is utilized to concurrently compute the k-list for each k ranging from 0 to $k_{\max}$ (lines 2-3). Based on our prior analysis, the work for generating the k-list is bounded by $O(l_{max} \cdot n + m)$. Consequently, the overall complexity of Algorithm 3 is $O(k_{max}(l_{max} \cdot n + m))$. It should be noted that, for a given directed graph, both $k_{\max}$ and $l_{\max}$ are upper-bounded by $O(\sqrt{m})$, where m represents the number of edges [16]. However, in practical scenarios, the values of k_{max} and l_{max} tend to be significantly smaller than $O(\sqrt{m})$. The span of Algorithm 3 is $O(k_{max} + k_{max} \cdot l_{max}) = O(k_{max}^2)$.

Practical optimization. During k-list computation, continuous scanning of vertices is necessary to identify those at the current level, incurring a time complexity of $O(l_{max} \cdot n)$. Graphs with a skewed distribution of out-degrees can substantially increase scanning overhead. To mitigate this, we propose a graph reduction strategy. When a fraction f of vertices in the graph has been peeled, we construct a new graph from the remaining vertices, preserving edges (u, v) if both u and v have an out-degree greater than l and an in-degree not less than k. This significantly reduces scanning overhead during subsequent levels, with scan time decreasing to $O((l_{\max} - l)(1 - f)n)$. The overall improvement depends on the values of l and f, aiming to minimize l and maximize f (as close to 1 as possible). The choice of f also determines the value of l. At a fixed level equal to a specific l, a certain percentage of vertices with a value of f will be removed from the graph. In this study, we set f to 0.98, following literature [26, 44]. Once the reduction process is completed, resulting out-core numbers are copied into corresponding vertices' out-degrees for the final outcome.

5 A SHELL-BASED PARALLEL D-CORE DECOMPOSITION ALGORITHM

In this section, we focus on optimizing our parallel D-core decomposition algorithm to improve its efficiency.

5.1 Shell-Based pruning techniques

During the D-core decomposition process, it is necessary to compute all the k-lists from 0 to k_{max}. However, in many real-world graphs, the k-lists corresponding to different values of k are identical. For example, there are 4 k-lists in the directed graph G in Figure 1. Notably, for k = 0 to 2, the k-lists correspond to the same D-cores. That is, the first three columns of Table 2 are identical. Thus, computing each k-list corresponding to all possible k would result in a significant amount of computational redundancy.

To minimize redundant computations by identifying identical k-lists with the same k value, we introduce the concept of (k, l)-shell, also known as D-shell.

Definition 4 ((k, l)-shell). *The (k, l)-shell of a directed graph G represents a set of vertices, where each vertex has an in-core number precisely equal to k for a given l.*

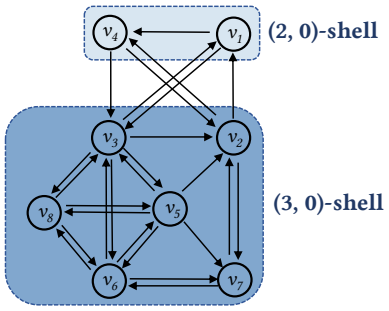


Figure 5: The (k, l)-shells of G in Figure 1 (l = 0).

Example 5. *Revisiting Example 1, we specifically focus on the scenario where $l = 0$ in graph G. As a result, we can observe the presence of two $(k, 0)$-shells, as shown in Figure 5: the first is the (2, 0)-shell, represented as $\{v_1, v_4\}$, and the second is the (3, 0)-shell, denoted as*

$\{v_2, v_3, v_5, v_6, v_7, v_8\}$. *The identification of these* $(k, 0)$*-shells is based on their having identical in-core numbers for the given value of* l.

When $l = 0$, the $(k, 0)$-shell includes all vertices within the $(k, 0)$-core with an in-core number exactly equal to k. Based on Definition 4, we can deduce the following lemma.

Lemma 5.1. *Given two integers* $k_i < k_j$*, their corresponding* k*-lists differ if: (1)* k_i *and* k_j *correspond to two different* $(k, 0)$*-shells; or (2) there exists a vertex* v *in the* k_i*-list such that:* $|\{u \in N^+(v) | L(k_i, u) \geq L(k_i, v)\}| < k_j$.

Proof. For case (1), it directly holds since the k_i-list contains vertices that do not belong to the k_j-list. For case (2), although k_i and k_j correspond to the same $(k, 0)$-shell, during the computation of the corresponding k-lists, the sequence of vertex removal is different because the removal order of v has changed. □

Based on Lemma 5.1, in D-core decomposition, we compute distinct k-lists to prevent redundancy. Besides, we exclude vertices within $(k, 0)$-shells with $k < k_i$ from computing k_i-list to enhance the process efficiency.

5.2 Improved k-list computation

The D-shell not only reduces redundant computations between k-lists but also can be used to accelerate the computation of each k-list. According to Definition 4, for a given value of k and l, we have the (k, l)-shell. This refers to the set of all vertices in the k-list whose out-core number equals l.

During k-list computation, we determine the out-core number of each vertex for a given value of k. Let's consider a vertex v with an out-degree of $d^-(v)$ and a given k. We assume that v's out-core number is $L(k, v)$. In Algorithm 2, while computing v's out-core number, v can be scanned at most $d^-(v) - L(k, v)$ times. Specifically, during the process from $level = L(k, v) - 1$ to $level = L(k, v)$, v's out-degree can be updated at most $d^-(v) - L(k, v)$ times before obtaining its out-core number. This is because in the worst-case scenario, v is one of the last vertices to be processed in the $(k, L(k, v))$-shell, as its out-degree only matches its out-core number when all its neighbors are removed. To reduce the number of scans for v and accelerate its out-core number calculation, we now discuss the relationship of (k, l)-shell between different k-lists by the following lemma.

Lemma 5.2. *Given two integers* k_1, k_2*, where* $k_1 < k_2$*, then for the same vertex* v *in the corresponding* k*-lists, the out-core number of* v *satisfies* $L(k_1, v) \geq L(k_2, v)$.

Proof. Let the in-core number of vertex v be denoted as k. We consider three distinct cases: $k < k_1, k_2 > k \geq k_1$, and $k \geq k_2$. In the case where $k < k_1$, it follows that in both the k_1-list and k_2-list, the out-core number of vertex v is 0. In the case where $k_2 > k \geq k_1$, we observe that in the k_1-list, $L(k_1, v) \geq 0$, while in the k_2-list, the out-core number $l_{k_2}(v)$ is 0. In the case where $k \geq k_2$, due to the fact that $k_1 < k_2$, the $(k_2, 0)$-core is contained in the $(k_1, 0)$-core. This implies that the $(k_2, 0)$-core is either equal to the $(k_1, 0)$-core or obtained by removing vertices from the vertex set of the $(k_1, 0)$-core. In the former scenario, the out-core numbers of vertices in both cores are equal. In the latter scenario, we establish that $L(k_1, v) \geq L(k_2, v)$, as the removal of vertices may cause a decrease in v's out-degree, thereby reducing its out-core number. □

Based on Lemma 5.2, it can be deduced that the out-core number of vertices is non-decreasing as k increases. In other words, for k_1-list and k_2-list where $k_1 > k_2$, the out-core number of vertices in the k_2-list is an upper bound for the corresponding vertices in the k_1-list. Building upon this observation, during the computation of the k-list, if we have already obtained k-lists for smaller values of k, we can utilize the previously derived out-core numbers of vertices as upper bounds for the current set of vertices, and then refine the upper bounds to the exact out-core numbers corresponding to the current value of k.

Presently, we introduce an efficient algorithm for computing out-core numbers of vertices at the current k value by leveraging previously obtained out-core numbers for lower k values. We denote k_p as the prior k value and k_c as the current k value. Notably, deriving the out-core numbers for all vertices corresponding to k_c based on their out-core numbers corresponding to k_p fundamentally entails a reordering of the out-core numbers of all vertices. As k increases, vertices originally within the $(k_p, 0)$-shell, which had values less than k_c, are removed, potentially leading to changes in the out-core numbers of their neighbors. Therefore, we enhance vertex out-core numbers by monitoring alterations in neighbor information, as opposed to repetitively peeling vertices and updating neighbor degrees, thereby avoiding redundant computations. For the out-core number of a vertex v at the current k value, i.e., $L(k, v)$, the following properties hold.

Property 2. *For an integer* k *and a vertex* v*, the out-core number* $L(k, v)$ *must satisfy the following conditions:*

(1) *there must be at least* $L(k, v)$ *out-neighbors of* v *whose out-core number at* k *is greater than or equal to* $L(k, v)$*;*
(2) *there must be at least* k *in-neighbors of* v *with an out-core number at* k *greater than or equal to* k.

That is,

$$\begin{cases} \left|\{u \in N^-(v) | L(k, u) \geq L(k, v)\}\right| \geq L(k, v), \\ \left|\{u \in N^+(v) | L(k, u) \geq L(k, v)\}\right| \geq k. \end{cases} \tag{1}$$

Example 6. *Consider the directed graph illustrated in Figure 1 as an example. Assuming we have already computed the 2-list, containing the out-core numbers of all vertices at* $k = 2$*, with* v_1, v_2, v_4*, and* v_7 *having* $L(2, v) = 2$*, and* v_3, v_5, v_6*, and* v_8 *having* $L(2, v) = 3$*. For vertex* v_3*, the out-core number* $L(2, v_3) = 3$ *implies that* v_3 *must have at least 3 out-neighbors with* $L(2, v)$ *greater than or equal to 3, specifically involving vertices* v_5*,* v_6*, and* v_8*. Meanwhile,* v_3 *should also have at least 2 in-neighbors with their* $L(2, v)$ *values greater than or equal to 2, which include vertices* v_1, v_4, v_5, v_6*, and* v_8.

Following Property 2, when computing the out-core numbers corresponding to the current k, we iteratively revise a vertex's out-core number by utilizing its neighbor information until convergence is attained, signifying that the out-core number remains constant. Specifically, we initialize the out-core numbers of all vertices at k_c with the out-core numbers of all vertices at the previous k_p. Subsequently, we exclude vertices in the $(k_p, 0)$-shell and refine the out-core numbers of the remaining vertices based on Property 2.

The pseudocode is listed in Algorithm 4. We introduce variable F to assess the convergence of vertex out-core numbers; V_c represents all vertices belonging to the $(k, 0)$-shells with k values greater than or equal to the current k_c (line 1); the boolean array *change* tracks

changes in the out-core numbers of vertices in V_c, initialized with all elements set to true. (line 2). For each vertex v within V_c, we initialize $L(k_c, v)$ to the previous out-core number $L(k_p, v)$ (lines 3-4). For a vertex v within V_c, when $change[v]$ is true, we proceed to find two values, t_1 and t_2, as follows. First, we identify t_1 as the maximum value from the out-core number set of all v's out-neighbors that satisfy case (1) of Property 2, which means t_1 is the maximum value of t for which $\left|u \in N^-(v) | L(k_c, u) \geq t\right| \geq t$ holds true (line 10). Next, we determine t_2 as the k_c-th largest element in the set $\{u \in N^+(v) | L(k_c, u)\}$, in order to fulfill case (2) of Property 2 (line 11). Subsequently, we compare $L(k_c, v)$ with the minimum value between t_1 and t_2. If $L(k_c, v)$ surpasses this minimum, we set the change status of its neighbors with out-core numbers larger than $min(t_1, t_2)$ and not exceeding $L(k_c, v)$ to $true$, as derived from Property 2. Then we set F to true and update $L(k_c, v)$ (lines 12-16). If none of the vertices in V_c have experienced changes in their out-core numbers, the process is concluded (line 5). Afterward, we determine the value of k_n that may result in different k-list compared to k_c, based on Lemma 5.1 (line 17).

Algorithm 4: PKL

Input: A directed graph $G = (V, E)$, k_p-list of G, and an integer $k_c > k_p$
Output: The k_c-list of G and an integer k_n

1 $F \leftarrow true$, $V_c \leftarrow$ vertices in $(k, 0)$-shells with $k \geq k_c$;
2 initialize $change$ of size $|V_c|$ with all elements set to $true$;
3 **for** $v \in V_c$ **do**
4 $L(k_c, v) \leftarrow L(k_p, v)$;
5 **while** F **do**
6 $F \leftarrow false$;
7 **for** $v \in V_c$ **in parallel do**
8 **if** $change[v]$ **then**
9 $change[v] \leftarrow false$;
10 $t_1 \leftarrow$ the maximum value of t satisfy $\left|\{u \in N^-(v) | L(k_c, u) \geq t\}\right| \geq t$;
11 $t_2 \leftarrow$ the k_c-th largest element in $\{u \in N^+(v) | L(k_c, u)\}$;
12 **if** $L(k_c, v) > min(t_1, t_2)$ **then**
13 **for** $u \in N(v)$ **do**
14 **if** $L(k_c, v) \geq L(k_c, u) > min(t_1, t_2)$ **then**
15 $change[u] \leftarrow true, F \leftarrow true$;
16 $L(k_c, v) \leftarrow min(t_1, t_2)$;
17 $k_n \leftarrow min_{v \in V_c}(\left|\{u \in N^+(v) | L(k_c, u) \geq L(k_c, v)\}\right|)$;
18 **return** $\{v \in V | L(k_c, v)\}, k_n$;

Time complexity. In Algorithm 4, updating the out-core number of vertex v takes $O(d(v))$ time. Thus, the time complexity for refining the out-core numbers of all vertices in V_c is $O(m_c)$, where m_c represents the number of edges in the induced subgraph of V_c. Consequently, the overall time complexity of the algorithm is $O(\sqrt{m_c} \cdot m_c)$, with $O(\sqrt{m_c})$ denoting the upper bound for the iterations of the while-loop in Algorithm 4.

Example 7. *In Figure 1, for $k = 2$, the out-core numbers for vertices v_1, v_2, v_4, and v_7 are 2, while for vertices v_3, v_5, v_6, and v_8, the out-core number is 3. As depicted in Figure 6, when $k = 3$, given that v_1 and v_4 are not part of the $(3, 0)$-shell, the out-core numbers of v_3 and v_2 associated with $k = 3$ may be altered. For v_3, we observe that $t_1 = 3$ and $t_2 = 3$, resulting in an unchanged out-core number. However, for v_2, the values of t_1 and t_2 are 1 and 2 respectively, prompting us to update $L(3, v_2)$ to 1 and mark the changing status of its neighbors as true. Then we find that v_7 has $t_1 = 1$ and $t_2 = 1$, so we update $L(3, v_7)$ as 1. Subsequently, we find that the out-core numbers of all vertices no longer change, signifying the termination of the entire process. Consequently, we derive the out-core numbers for all the vertices corresponding to $k = 3$. It's worth noting that the entire computation involves only a subset of the graph, thus avoiding the need to scan all vertices and sequentially remove them level by level.*

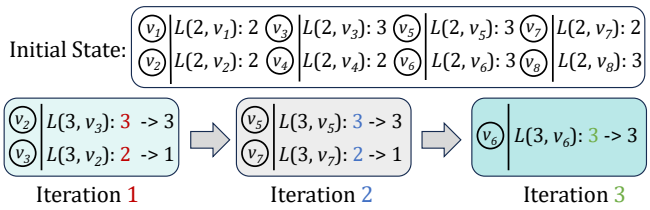


Figure 6: Process of computing 3-list of G in Figure 1.

5.3 Overall algorithm

Algorithm 5 presents the overall algorithm for shell-based parallel D-core decomposition (Shell-PDC). The approach involves an initial computation of all $(k, 0)$-shells to determine all possible values of k. It also calculates all $(0, l)$-shells to establish the out-core number of all vertices when $k = 0$, denoted as $L(0, v)$, serving as the initial value for each vertex's out-core number. The out-core numbers of all vertices at $k = 0$ are collected into both $temp$ and Res, with Res serving as a repository for all k-lists (line 1). Next, $\mathcal{K}$ is computed, containing distinct k values associated with various $(k, 0)$-shells (line 2). For each k in $\mathcal{K}$, Algorithm 4 is invoked, iteratively computing the k-list based on previous out-core numbers of vertices and appending them to Res (lines 3-5). $k_n + 1$ signifies a potential value with a distinct k-list compared to the current k. If it is less than the next element in $\mathcal{K}$, it is inserted before that element for subsequent computation (lines 6-7).

Algorithm 5: Shell-PDC

Input: A directed graph $G = (V, E)$
Output: All the k-lists of G

1 $temp \leftarrow$ compute the 0-list of G, $Res \leftarrow temp$; // Alogorithm 2
2 $\mathcal{K} \leftarrow$ the set of k-values for each $(k, 0)$-shell;
3 **for** $k \in \mathcal{K}$ *in ascending order* **do**
4 $temp, k_n \leftarrow$ PKL$(k, temp)$; // Alogorithm 4
5 $Res \leftarrow Res \cup temp$;
6 **if** $k_n + 1 < \mathcal{K}.next(k)$ **then**
7 insert $(k_n + 1)$ into $\mathcal{K}$ before $\mathcal{K}.next(k)$;
8 **return** Res ;

Analysis. Algorithm 2 requires $O(l_{max} \cdot n + m)$ time to obtain the 0-list of G, and it takes $O(k_{max} \cdot n + m)$ to compute all the $(k, 0)$-shells of G. For a given k_i, Algorithm 4 takes $O(\sqrt{m_i} \cdot m_i)$

time to compute the k_i-list, where m_i is the number of edges in the subgraph of G induced by the $(k_i, 0)$-core. Consequently, computing all the k-lists with $k > 0$ takes $O(\sum_{i=1}^{k_{max}} \sqrt{m_i} \cdot m_i)$ in the worst-case scenario. In summary, the overall time complexity of Algorithm 5 is $O(\sum_{i=1}^{k_{max}} \sqrt{m_i} \cdot m_i + (k_{max} + l_{max}) \cdot n + m)$. In practice, k_{max} and l_{max} are considerably smaller than the maximum out-degree and maximum in-degree. Additionally, the values of m_i decrease significantly as k_i increases. The span of `Shell-PDC` is $O(k_{max}^2)$.

6 EXPERIMENTS

We now present the experimental results. Section 6.1 discusses the setup. We report the efficiency results in Sections 6.2 and 6.3.

6.1 Setup

Datasets. We use ten real large directed graphs. Specifically, Email-EuAll is a communication network generated using email data, Amazon is an e-commerce graph; Pokec, Live Journal, and Slashdot are social networks; these five graphs are obtained from SNAP[2]. Hollywood is an actors collaboration graph; Enwiki-2013 is an encyclopedia graph generated from the English part of Wikipedia; Webbase, IT-2004, and UK-2007 are web graphs; these five graphs are sourced from LAW[3]. Table 4 reports the statistics of each graph, where n is the number of vertices, m is the number of edges, k_{max} and l_{max} are the maximum in-core and out-core numbers respectively.

Table 4: Directed graphs used in the experiments.

Graphs	Abbr.	Category	n	m	k_{max}	l_{max}
Email-EuAll	EM	Communication	265.21K	418.96K	27	27
Slashdot	SD	Social	82.17K	870.16K	53	53
Amazon	AM	Product	400.73K	3.20M	10	10
Pokec	PO	Social	1.63M	30.62M	32	31
Live Journal	LJ	Social	4.85M	68.48M	252	252
Enwiki-2013	EW	Text	4.21M	101.31M	89	107
Hollywood	HW	Actors	2.18M	228.99M	1,297	1,297
Webbase	WB	Hyperlink	118.14M	992.84M	1,218	1,218
IT-2004	IT	Web	41.29M	1.14B	3,198	3,198
UK-2007	UK	Web	978.41M	3.92B	10,027	10,027

Algorithms. We test the following D-core decomposition algorithms:

- **`Peeling` [16]:** the state-of-the-art sequential D-core decomposition algorithm;
- **`AC` [30]:** the distributed anchored coreness-based D-core decomposition algorithm, and we parallelize it by using multi-threads;
- **`SC` [30]:** the state-of-the-art distributed skyline coreness-based D-core decomposition algorithm, and we parallelize it by using multi-threads;
- **`ParPeel`:** our proposed parallel D-core decomposition algorithm, which is depicted in Algorithm 3;
- **`ParPeel+`:** our proposed parallel D-core decomposition algorithm in Algorithm 3 with pruning strategy in Lemma 3 of [16];
- **`Shell-PDC`:** our proposed parallel D-core decomposition algorithm, listed in Algorithm 5.

Experimental settings. In the experiments, we implement all the aforementioned algorithms in C++ and compile them with the gcc 9.4.0 compiler using the -O3 optimization level. For all parallel algorithms, we base the implementations on OpenMP, a widely adopted shared-memory programming interface that supports symmetrical multi-processing architectures, such as multi-core CPUs. The experiments are run on a Linux machine running Ubuntu Linux 20.04.5 LTS. This machine is equipped with dual Intel Xeon(R) Gold 6338 2.0GHz processors (32 cores) and 496GB of RAM. The number of threads p varies from 1 to 32, and we set p=32 by default. We teminate the algorithm's execution when the running time exceeds 10^8 ms (10^5 s) and mark it as **INF**.

6.2 Efficiency evaluation

• **Overall efficiency results.** We evaluate the runtime of all parallel D-core decomposition algorithms across all datasets using 32 threads. As `peeling` is challenging to parallelize, we execute it using a single thread. The results are listed in Figure 7.

Clearly, our `Shell-PDC` algorithm exhibits superior efficiency across all datasets, particularly excelling on large-scale datasets with over a billion edges, such as IT and UK. To elaborate, `Shell-PDC` demonstrates remarkable speed, being three orders of magnitude faster than the anchored coreness algorithm `AC` on smaller datasets like EM and SD. While the state-of-the-art parallel D-core decomposition algorithm `SC` outperforms `AC`, our approach remains significantly faster, achieving up to three orders of magnitude improvement on larger datasets like IT, where the runtime of `SC` exceeds 10^5 seconds. For `ParPeel` and `ParPeel+`, they showcase superior performance compared to `AC` and `SC`, while our algorithm still outperforms them by up to two orders of magnitude. The primary reason behind this is that `Shell-PDC` strategically avoids scanning all vertices in the graph during each iteration, effectively reducing redundant computations and minimizing excessive synchronization overhead.

`ParPeel+` is slower than `ParPeel`. This is primarily due to two reasons: 1) the pruning effectiveness of Lemma 3 in [16] is limited in datasets where the values of k_{max} and the number of $(k, 0)$-shells remained consistent (e.g., EM); 2) the process of determining whether the k-list corresponding to the current k needs computation, as per Lemma 3, incurred additional computational overhead that is challenging to parallelize. It's noteworthy that in extensive datasets like WB, IT, and UK, both `SC` and `ParPeel` encounter out-of-memory (OOM) issues. This is due to the algorithms requiring substantial storage for the out-core numbers of each vertex, leading to high space occupation, especially when the vertex degree is large.

While `Peeling` is a sequential algorithm, its efficiency closely rivals that of the parallel algorithm `SC` with 32 threads. This is primarily attributed to optimizations like binsort [3], leading to an $O(m)$ time complexity for each computation of the k-list. Nevertheless, our `Shell-PDC` algorithm can still achieve a speedup of up to two orders of magnitude compared to `Peeling`.

• **Effect of the number of threads.** In Figure 8, the efficiency of all parallel D-core decomposition algorithms is depicted as the number of threads ranges from 1 to 32 across all datasets. With an increasing number of threads, the runtime of `Shell-PDC` exhibits

[2]https://snap.stanford.edu/data/index.html
[3]https://law.di.unimi.it/index.php

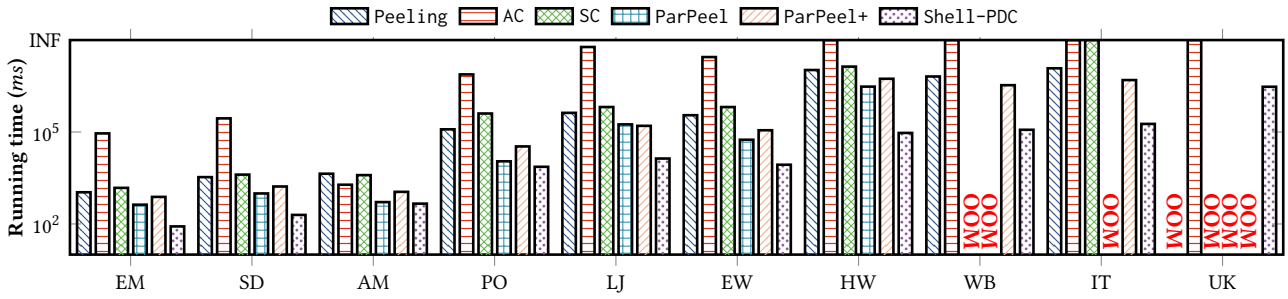


Figure 7: Efficiency of all D-core decomposition algorithms on all datasets.

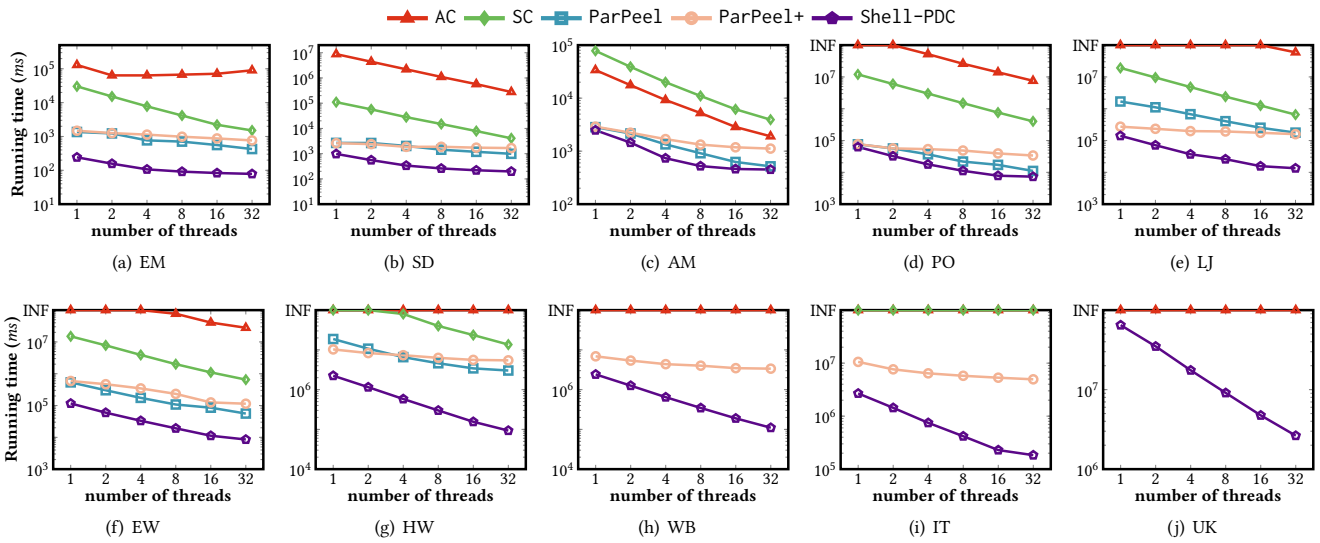


Figure 8: Effect of the number of threads.

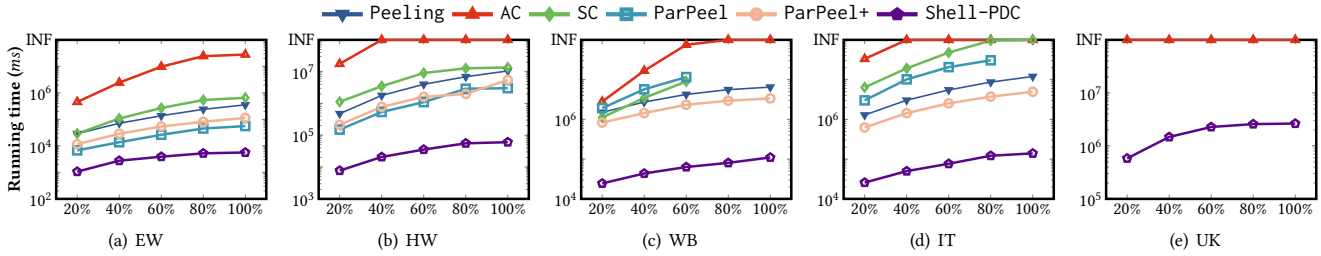
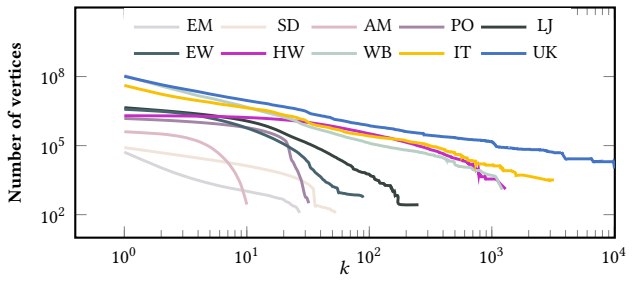


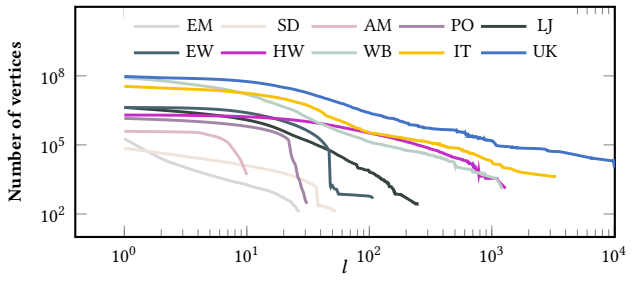
Figure 9: Scalability test.

a linear reduction, showcasing strong parallel scalability. Particularly, employing 32 threads enables our Shell-PDC to achieve a self speedup of 24.57 times. However, for ParPeel and ParPeel+, the self speedup ratio is limited due to the continual reduction in the graph's size during k-list computations. Although this reduction decreases the workload for each k-list computation, the independent nature of these computations necessitates frequent thread activation and termination, resulting in increased overhead and impacting their parallel performance. While the self speedup ratio for AC and SC may surpass that of ParPeel and ParPeel+, their inherently high time complexity leads to their runtime being 89.90 times slower than that of Shell-PDC.

• **Scalability test.** For scalability testing, we randomly select 20%, 40%, 60%, 80%, and 100% of edges from each dataset, creating five subgraphs induced by these respective edge percentages. Due to limited space, we present results solely for the six graphs with the largest edge counts, as similar trends prevail across other datasets. As depicted in Figure 9, the time costs for all algorithms increase



(a) $(k, 0)$-core



(b) $(0, l)$-core

Figure 10: Trends of vertex counts of $(k, 0)$-core and $(0, l)$-core on all datasets.

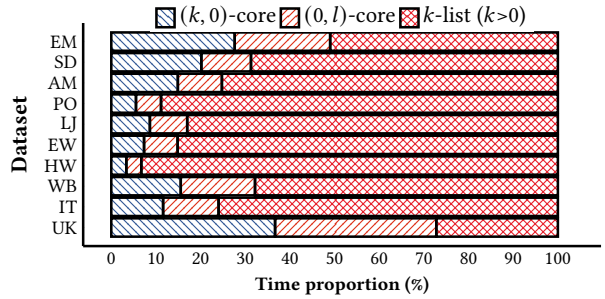
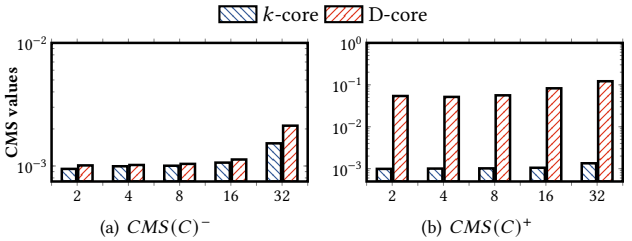


Figure 11: Proportion of time cost of each step in `Shell-PDC`.



(a) $CMS(C)^-$ (b) $CMS(C)^+$

Figure 12: Community comparison of community member similarity (CMS).

as the dataset size expands. Notably, among the algorithms, our `Shell-PDC` demonstrates the least variability in its performance changes, highlighting its superior scalability. Conversely, AC and SC exhibit relatively efficient performance with smaller datasets. However, as the graph size grows, their runtimes notably increase due to these algorithms' heightened sensitivity to vertex degrees.

6.3 Additional evaluations

• **Comparing the sizes of the graphs processed.** Table 5 summarizes the count of k-lists computed by `Shell-PDC` alongside the number of $(k, 0)$-shells across each dataset. Notably, the number of $(k, 0)$-shells is considerably lower than k_{max} in most datasets. This occurrence primarily results from the distribution of degrees in real-world graphs, often following a power-law distribution. Consequently, these graphs typically exhibit a dense region, which contains a significant gap between sparse areas. The number of k-lists computed by `Shell-PDC` closely matches the count of $(k, 0)$-shells, particularly evident in the first four datasets.

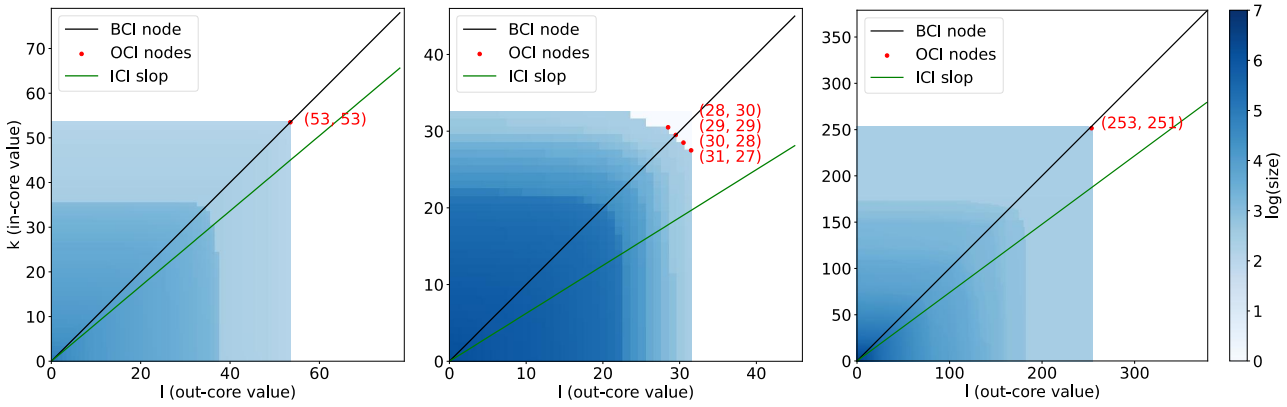
Figure 10(a) presents the count of vertices within each k-list across all datasets. Note that the number of vertices in the subgraph corresponding to a k-list between two adjacent $(k, 0)$-shells is identical. Evidently, the graph size processed by `Shell-PDC` notably decreases as k increases. Particularly, when k approaches k_{max}, the dense region of the graph significantly reduces in size. We also compare the trends in vertex counts of $(0, l)$-cores in Figure 10(b). The results indicate that the trends in vertex counts of $(0, l)$-cores across datasets resemble those of $(k, 0)$-cores. Therefore, if we calculate $l_{\max}$ initially and subsequently compute D-cores corresponding to each l, the efficiency of the decomposition process mirrors that of computing D-cores corresponding to each k.

Table 5: The number of $(k, 0)$-shells and k-lists computed by `Shell-PDC`.

datasets	EM	SD	AM	PO	LJ
# of $(k, 0)$-shells	28	53	11	33	173
`Shell-PDC`	28	54	11	33	177
datasets	EW	HW	WB	IT	UK
# of $(k, 0)$-shells	89	821	353	544	697
`Shell-PDC`	90	841	425	686	901

• **Time cost of different steps in `Shell-PDC`.** The runtime of `Shell-PDC` comprises three parts: (1) computing all the $(k, 0)$-shells in the graph by setting $l = 0$, essentially computing G's $(k, 0)$-core; (2) setting $k = 0$ and computing the value of $L(0, v)$ for each vertex v, i.e., calculating G's $(0, l)$-core; and (3) initiating $L(0, v)$ as the initial out-core number for each vertex and iteratively calculating $L(k, v)$ until all k-lists are obtained. Figure 11 illustrates the proportion of time taken for each phase across all datasets. Notably, the durations of phases (1) and (2) are similar and collectively account for a smaller portion of the total time compared to the iterative computation of k-lists with $k > 0$. However, in the UK dataset, the time spent in phase (3) is less than that in phases (1) and (2). This is primarily influenced by the distribution of vertex degrees in the graph. In the dataset, the initial out-core numbers obtained in phase (2) closely align with the final values in each k-list. Consequently, the computation in phase (3) is relatively fast.

• **Convergence evaluation.** In this experiment, we assess the number of iterations required for our `Shell-PDC` algorithm during the computation of k-lists with $k > 0$. Table 6 presents the results across all datasets. It's noteworthy that the observed number of iterations is notably lower than the upper bound, represented by the maximum degree among all vertices. It's important to note



(a) SD (b) PO (c) LJ

Figure 13: The distribution of (k, l) values of the D-core decomposition of SD, PO, and LJ.

that on select datasets, the iteration count grows, influenced by vertex degrees and connected to the graph's topological structure. Generally, the trend indicates that larger graphs tend to require a higher iteration count due to an increase in the number of $(k, 0)$-shells within these graphs.

Table 6: The number of iterations required for `Shell-PDC` to compute all k-list with $k > 0$.

datasets	EM	SD	AM	PO	LJ
# iterations	147	384	119	1,476	3,163
d_{max}	7,631	2,552	2,747	13,733	20,292
datasets	EW	HW	WB	IT	UK
# iterations	1,895	840	3,607	4,866	10,166
d_{max}	431,795	13,107	816,127	1,326,744	1,261,714

• **Case studies.** In this experiment, we conduct two case studies focusing on the practical applications of D-core decomposition: directed graph collaboration analysis and community search.

Directed graph collaboration analysis. D-core decomposition is instrumental in understanding collaboration patterns within directed graphs [19], where dense in/outlink connectivity signifies collaborative tendencies. Giatsidis et al. [21] introduced several metrics to assess collaboration and robustness, including the Balanced Collaboration Index (BCI), Optimal Collaboration Index (OCI), Inherent Collaboration Index (ICI), and Average Collaboration Index (ACI). The first three metrics correspond to specific D-cores within the directed graph, while the ACI represents half of the average Manhattan distance of the graph's frontier cells. Here, the frontier cells correspond to (k, l) pairs of each D-core in the directed graph. In this experiment, we apply D-core decomposition to three social networks (i.e., SD, PO, and LJ) and visualize the D-core matrices in Figure 13. We observe that BCI, OCI, and ICI correspond to specific D-cores within the datasets, representing robust regions of the graphs. The details of the collaboration indices are listed in Table 7.

Table 7: Collaboration indices.

Graphs	SD	PO	LJ
BCI (k, l)	(53, 53)	(29, 29)	(251, 251)
OCI (k, l)	(53, 53)	(31, 27), (29, 29), (30, 28), (28, 30)	(253, 251)
ICI (k, l)	1.013	0.871	0.934
ACI	23.873	40	189.625

Community search. In this experiment, we conduct community search on the LJ and compare the results with the k-core-based community [43]. We utilize the community member similarity (CMS) to measure similarities for the community H on directed graphs [16], which is defined as

$$CMS(H)^- = \frac{1}{|H|^2} \sum_{u \in H} \sum_{v \in H} \frac{d^-(u) \cap d^-(v)}{d^-(u) \cup d^-(v)},$$
$$CMS(H)^+ = \frac{1}{|H|^2} \sum_{u \in H} \sum_{v \in H} \frac{d^+(u) \cap d^+(v)}{d^+(u) \cup d^+(v)}. \quad (2)$$

We randomly select a query vertex q and compute a D-core community containing q with $k = l$. For comparison, we ignore the directions of edges and compute the k'-core community containing q with $k' = k + l$. We then vary the values of k and l from $\{2, 4, 8, 16, 32\}$ to obtain the communities and calculate the CMS of the two community models. The results are presented in Figure 12. It's important to note that the D-core community exhibits higher CMS compared to the k-core community across all parameters. This observation suggests that leveraging directional information enables us to obtain more cohesive communities.

7 CONCLUSION

In this paper, we investigate the problem of parallel D-core decomposition over large directed graphs, by harnessing the computational power of multicore CPUs. We first develop a parallel D-core decomposition algorithm that strategically computes D-cores for each conceivable k, employing an implicit level-by-level vertex removal strategy. It not only diminishes computational dependencies, but also maintains a favorable time complexity, comparable to that

of sequential algorithms. We further propose a shell-based parallel D-core decomposition algorithm, by introducing a novel concept of D-shell which allows us reduce redundant computations during the decomposition process, and deriving D-cores with larger k from the calculated D-cores based on D-shell, aiming to improve both efficiency and parallelism. We have conducted extensive experiments on ten real-world large graphs. The experimental results show that our algorithms are highly efficient and scalable, and our shell-based decomposition algorithm outperforms state-of-the-art parallel algorithms by up to two orders of magnitude with 32 threads.

In the future work, an interesting research direction is to extend our algorithms for the setting with heterogeneous hardware, such as FPGA and GPU, which may achieve even higher speedup.

REFERENCES

[1] J Alvarez-Hamelin, Luca Dall'Asta, Alain Barrat, and Alessandro Vespignani. 2005. Large scale networks fingerprinting and visualization using the k-core decomposition. *Advances in neural information processing systems* 18 (2005).

[2] 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.

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

[4] Francesco Bonchi, Francesco Gullo, Andreas Kaltenbrunner, and Yana Volkovich. 2014. Core decomposition of uncertain graphs. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 1316–1325.

[5] Richard P Brent. 1974. The parallel evaluation of general arithmetic expressions. *Journal of the ACM (JACM)* 21, 2 (1974), 201–206.

[6] Lijun Chang and Lu Qin. 2019. Cohesive Subgraph Computation Over Large Sparse Graphs. In *35th IEEE International Conference on Data Engineering, ICDE 2019*. IEEE, 2068–2071.

[7] Lijun Chang, Jeffrey Xu Yu, Lu Qin, Xuemin Lin, Chengfei Liu, and Weifa Liang. 2013. Efficiently computing k-edge connected components via graph decomposition. In *Proceedings of the 2013 ACM SIGMOD international conference on management of data*. 205–216.

[8] Yankai Chen, Jie Zhang, Yixiang Fang, Xin Cao, and Irwin King. 2021. Efficient community search over large directed graphs: An augmented index-based approach. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*. 3544–3550.

[9] 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*. IEEE, 51–62.

[10] Qiangqiang Dai, Rong-Hua Li, Meihao Liao, Hongzhi Chen, and Guoren Wang. 2022. Fast maximal clique enumeration on uncertain graphs: A pivot-based approach. In *Proceedings of the 2022 International Conference on Management of Data*. 2034–2047.

[11] Maximilien Danisch, Oana Balalau, and Mauro Sozio. 2018. Listing k-cliques in sparse real-world graphs. In *Proceedings of the 2018 World Wide Web Conference*. 589–598.

[12] 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 (Big Data)*. IEEE, 9–16.

[13] Hossein Esfandiari, Silvio Lattanzi, and Vahab Mirrokni. 2018. Parallel and streaming algorithms for k-core decomposition. In *international conference on machine learning*. PMLR, 1397–1406.

[14] Yixiang Fang, Wensheng Luo, and Chenhao Ma. 2022. Densest subgraph discovery on large graphs: Applications, challenges, and techniques. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3766–3769.

[15] Yixiang Fang, Kai Wang, Xuemin Lin, and Wenjie Zhang. 2021. Cohesive subgraph search over big heterogeneous information networks: Applications, challenges, and solutions. In *Proceedings of the 2021 International Conference on Management of Data*. 2829–2838.

[16] 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.

[17] 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.

[18] Yixiang Fang, Kaiqiang Yu, Reynold Cheng, Laks VS Lakshmanan, and Xuemin Lin. 2019. Efficient algorithms for densest subgraph discovery. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1719–1732.

[19] 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.

[20] Loukas Georgiadis, Evangelos Kipouridis, Charis Papadopoulos, and Nikos Parotsidis. 2023. Faster computation of 3-edge-connected components in digraphs. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 2489–2531.

[21] 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.

[22] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-truss community in large and dynamic graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1311–1322.

[23] Xin Huang, Wei Lu, and Laks VS Lakshmanan. 2016. Truss decomposition of probabilistic graphs: Semantics and algorithms. In *Proceedings of the 2016 International Conference on Management of Data*. 77–90.

[24] Shweta Jain and C Seshadhri. 2020. The power of pivoting for exact clique counting. In *Proceedings of the 13th International Conference on Web Search and Data Mining*. 268–276.

[25] Joseph JáJá. 1992. *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc.

[26] Humayun Kabir and Kamesh Madduri. 2017. Parallel k-core decomposition on multicore platforms. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 1482–1491.

[27] 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.

[28] Ronghua Li, Sen Gao, Lu Qin, Guoren Wang, Weihua Yang, and Jeffrey Xu Yu. 2020. Ordering Heuristics for k-clique Listing. *Proc. VLDB Endow.* (2020).

[29] Rong-Hua Li, Qiangqiang Dai, Guoren Wang, Zhong Ming, Lu Qin, and Jeffrey Xu Yu. 2019. Improved algorithms for maximal clique search in uncertain networks. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1178–1189.

[30] Xuankun Liao, Qing Liu, Jiaxin Jiang, Xin Huang, Jianliang Xu, and Byron Choi. 2022. Distributed D-core decomposition over large directed graphs. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1546–1558.

[31] Boge Liu, Long Yuan, Xuemin Lin, Lu Qin, Wenjie Zhang, and Jingren Zhou. 2020. Efficient (α, β)-core computation in bipartite graphs. *The VLDB Journal* 29, 5 (2020), 1075–1099.

[32] 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 SIGMOD International Conference on Management of Data*. 2183–2197.

[33] 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), 10168.

[34] Wensheng Luo, Kenli Li, Xu Zhou, Yunjun Gao, and Keqin Li. 2022. Maximum Biplex Search over Bipartite Graphs. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 898–910.

[35] Wensheng Luo, Zhuo Tang, Yixiang Fang, Chenhao Ma, and Xu Zhou. 2023. Scalable algorithms for densest subgraph discovery. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 287–300.

[36] Wensheng Luo, Qiaoyuan Yang, Yixiang Fang, and Xu Zhou. 2023. Efficient Core Maintenance in Large Bipartite Graphs. *Proceedings of the ACM on Management of Data* 1, 3 (2023), 1–26.

[37] Bingqing Lyu, Lu Qin, Xuemin Lin, Ying Zhang, Zhengping Qian, and Jingren Zhou. 2020. Maximum biclique search at billion scale. *Proceedings of the VLDB Endowment* (2020).

[38] Chenhao Ma, Yixiang Fang, Reynold Cheng, Laks VS Lakshmanan, Wenjie Zhang, and Xuemin Lin. 2020. Efficient algorithms for densest subgraph discovery on large directed graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1051–1066.

[39] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. 2013. Distributed k-Core Decomposition. *IEEE Transactions on Parallel and Distributed Systems* 2, 24 (2013), 288–300.

[40] 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.

[41] Ahmet Erdem Sariyuce, C Seshadhri, and Ali Pinar. 2017. Local algorithms for hierarchical dense subgraph discovery. *arXiv preprint arXiv:1704.00386* (2017).

[42] Ahmet Erdem Sariyuce, C Seshadhri, Ali Pinar, and Umit V Catalyurek. 2015. Finding the hierarchy of dense subgraphs using nucleus decompositions. In *Proceedings of the 24th International Conference on World Wide Web*. 927–937.

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

[44] Jessica Shi, Laxman Dhulipala, and Julian Shun. 2021. Theoretically and practically efficient parallel nucleus decomposition. *Proceedings of the VLDB Endowment* 15, 3 (2021), 583–596.

[45] Henry Soldano, Guillaume Santini, Dominique Bouthinon, and Emmanuel Lazega. 2017. Hub-authority cores and attributed directed network mining. In *2017 IEEE 29th International conference on tools with artificial intelligence (ICTAI)*. IEEE, 1120–1127.

[46] Zitan Sun, Xin Huang, Jianliang Xu, and Francesco Bonchi. 2021. Efficient probabilistic truss indexing on uncertain graphs. In *Proceedings of the Web Conference 2021*. 354–366.

[47] Anxin Tian, Alexander Zhou, Yue Wang, and Lei Chen. 2023. Maximal D-truss search in dynamic directed graphs. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2199–2211.

[48] Jia Wang and James Cheng. 2012. Truss Decomposition in Massive Networks. *Proceedings of the VLDB Endowment* 5, 9 (2012).

[49] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2020. Efficient bitruss decomposition for large-scale bipartite graphs. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 661–672.

[50] Jianye Yang, Yun Peng, Dian Ouyang, Wenjie Zhang, Xuemin Lin, and Xiang Zhao. 2023. (p, q)-biclique counting and enumeration for large sparse bipartite graphs. *The VLDB Journal* (2023), 1–25.

[51] Michael Yu, Dong Wen, Lu Qin, Ying Zhang, Wenjie Zhang, and Xuemin Lin. 2021. On querying historical k-cores. *Proceedings of the VLDB Endowment* (2021).