Scaling Up Distance-generalized Core Decomposition

Qiangqiang Dai
Beijing Institute of Technology
Beijing, China
qiangd66@gmail.com

Rong-Hua Li
Beijing Institute of Technology
Beijing, China
lironghuascut@gmail.com

Lu Qin
University of Technology Sydney
Sydney, Australia
Lu.Qin@uts.edu.au

Guoren Wang
Beijing Institute of Technology
Beijing, China
wanggrbit@126.com

Weihua Yang
Taiyuan University of Technology
Taiyuan, China
yangweihua@tyut.edu.cn

Zhiwei Zhang and Ye Yuan
Beijing Institute of Technology
Beijing, China
cszwzhang@outlook.com
yuanye@mail.neu.edu.cn

ABSTRACT

Core decomposition is a fundamental operator in network analysis. In this paper, we study a problem of computing distance-generalized core decomposition on a network. A distance-generalized core, also termed (k, h)-core, is a maximal subgraph in which every vertex has at least k other vertices at distance no larger than h. The state-of-the-art algorithm for solving this problem is based on a peeling technique which iteratively removes the vertex (denoted by v) from the graph that has the smallest h-hop degree. The h-hop degree of a vertex v denotes the number of other vertices that are reachable from v within h hops. Such a peeling algorithm, however, needs to frequently recompute the h-hop degrees of v's neighbors after deleting v, which is typically very costly for a large h. To overcome this limitation, we propose an efficient peeling algorithm based on a novel h-hop degree updating technique. Instead of recomputing the h-hop degrees, our algorithm can dynamically maintain the h-hop degrees for all vertices via exploring a very small subgraph, after peeling a vertex. We show that such an h-hop degree updating procedure can be efficiently implemented by an elegant bitmap technique. In addition, we also propose a sampling-based algorithm and a parallelization technique to further improve the efficiency. Finally, we conduct extensive experiments on 12 real-world graphs to evaluate our algorithms. The results show that, when $h \geq 3$, our exact and sampling-based algorithms can achieve up to 10× and 100× speedup over the state-of-the-art algorithm, respectively.

CCS CONCEPTS

• **Theory of computation** → **Graph algorithms analysis**.

KEYWORDS

cohesive subgraph, core decomposition, distance-generalized core decomposition

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CIKM '21, November 1–5, 2021, Virtual Event, QLD, Australia
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8446-9/21/11…$15.00
https://doi.org/10.1145/3459637.3482294

ACM Reference Format:
Qiangqiang Dai, Rong-Hua Li, Lu Qin, Guoren Wang, Weihua Yang, and Zhiwei Zhang and Ye Yuan. 2021. Scaling Up Distance-generalized Core Decomposition. In *Proceedings of the 30th ACM International Conference on Information and Knowledge Management (CIKM '21), November 1–5, 2021, Virtual Event, QLD, Australia.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3459637.3482294

1 INTRODUCTION

Many real-world networks such as social networks, biological networks, and collaboration networks often contain cohesive subgraph structures. Finding cohesive subgraphs from a network is a fundamental problem in networks analysis which has attracted much attention in recent years [5, 7, 8, 10, 32]. A variety of cohesive subgraph models have been proposed, such as maximal clique [11, 12], k-plex [8, 29], k-truss [13, 20, 32], and k-core [28]. Among of them, k-core is the most appealing model, because it can be computed in linear time [6]. However, computing cohesive subgraphs based on the other models is often very costly. As a consequence, the k-core model has been widely used in many application domains, including community discovery [14, 19], network topology analysis [30], protein complex modeling [2, 4], and network visualization [3] [34].

The k-core of a graph G is defined as a maximal subgraph in which every vertex has a degree at least k within that subgraph. Although it is commonly used in practice, the k-core model sometimes cannot detect cohesive subgraphs. For example, let us consider a graph shown in Fig. 1. Intuitively, the subgraph induced by the vertices $\{v_8, v_9, \cdots, v_{14}\}$ is a cohesive subgraph. Such a cohesive subgraph, however, cannot be identified by the k-core model. This is because the entire graph is 2-core, and we cannot distinguish the cohesive subgraph and the entire graph based on different k values using the k-core model.

To overcome this limitation, Bonchi et al. [10] recently proposed a distance-generalized k-core concept, called (k, h)-core, where k and h ($h \geq 1$) are two integer parameters. Specifically, the (k, h)-core is a maximal subgraph in which every vertex has at least k other vertices with distance at most h within that subgraph. As indicated in [10], such a distance-generalized k-core model can detect cohesive subgraphs that cannot be found by the traditional k-core model. Reconsider the graph in Fig. 1. Suppose that $h = 2$. We can easily verify that the subgraph induced by $\{v_8, v_9, \cdots, v_{14}\}$ is a $(6, 2)$-core, while the entire graph is a $(4, 2)$-core. Therefore,

we are able to apply the (k,h)-core model to identify the cohesive subgraph induced by $\{v_8, v_9, \cdots, v_{14}\}$.

In this paper, we focus on the problem of computing all (k,h)-cores on a graph G for a given parameter h. Such a problem is also called (k,h)-core decomposition. The (k,h)-core decomposition has many applications in practice. As shown in [10], the (k,h)-core decomposition can be used to speed up the computation of finding the maximum h-club on a graph and find a good approximation for the distance-generalized densest subgraph problem.

To compute the (k,h)-core decomposition, Bonchi et al. [10] proposed a peeling algorithm which iteratively removes the vertex that has the smallest h-hop degree until all vertices are deleted. Here the h-hop degree of a vertex v is defined as the number of other vertices that are reachable from v within h hops. The defect of such a peeling algorithm is that it needs to recompute the h-hop degrees for all vertices in v's h-hop neighborhood when peeling a vertex v, which is often costly for a large h. Here the h-hop neighborhood of v, denoted by $N_v^h(G)$, is a set of other vertices that are reachable from v within h hops. Bonchi et al. [10] also developed an improved algorithm with several lower and upper bounding techniques to alleviate such h-hop degree re-computation costs. However, as shown in our experiments, such an improved peeling algorithm is still very costly for $h \geq 3$ on large graphs, because the algorithm may still need to frequently recompute the h-hop degrees.

To circumvent this issue, we propose an efficient peeling algorithm, called KHCore, based on a novel h-hop degree updating technique. Specifically, when peeling a vertex v, we prove that the h-hop degree for each vertex in $N_v^h(G)$ can be updated by exploring a small subgraph induced by $N_v^h(G)$. Based on this key result, we devise the KHCore algorithm which does not recompute the h-hop degrees for all vertices in $N_v^h(G)$, but it updates the h-hop degrees for every vertex in $N_v^h(G)$ by only accessing a small subgraph induced by $N_v^h(G)$, thus it is very efficient in practice. We also develop an elegant bitmap technique to implement the h-hop degree updating procedure which not only improves the efficiency, but it also reduces the space usage of our algorithm. In addition, a sampling-based algorithm is also presented to further improve the efficiency. To scale to larger graphs, we also propose a parallelization strategy to parallelize our algorithms for (k,h)-core decomposition. Finally, we conduct extensive experiments using 12 real-world datasets to evaluate the proposed algorithms. The results show that, if $h \geq 3$, our exact and sampling-based algorithms (with a sampling rate $r = 0.1$) using the bitmap technique can achieve up to 10× and 100× acceleration over the state-of-the-art algorithm. To summarize, the main contributions of this paper are as follows.

- **A new algorithm.** We propose a new peeling algorithm, called KHCore, for (k,h)-core decomposition. The appealing feature of KHCore is that it can update the h-hop degrees for all vertices in $N_v^h(G)$ when peeling a vertex v by exploring a small subgraph induced by $N_v^h(G)$, without recomputing the h-hop degrees for all vertices in $N_v^h(G)$.
- **Optimization techniques.** We develop a bitmap technique, a sampling-based algorithm, and a parallelization strategy to improve the efficiency and scalability of KHCore.
- **Extensive experiments.** We make use of 12 large real-world datasets to evaluate our algorithms, and the results demonstrate the efficiency and scalability of our algorithms. The source code is available at https://github.com/BITDataScience/khcore.

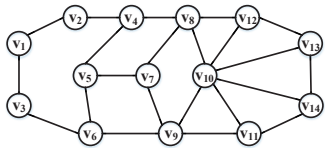

Figure 1: Running example

2 PROBLEM STATEMENT

In this paper, we focus on an undirected and unweighted graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. Let $n = |V|$ and $m = |E|$ be the number of vertices and edges respectively. For each vertex v, the neighborhood of v, denoted by $N_v(G)$, is defined as $N_v(G) \triangleq \{u \in V | (v,u) \in E\}$. The degree of a vertex v in G, denoted by $d_v(G)$, is the cardinality of $N_v(G)$, i.e., $d_v(G) = |N_v(G)|$. For simplicity, we use N_v and d_v to denote $d_v(G)$ and $N_v(G)$ respectively if the context is clear. Let $G(S) = (S, E(S))$ be an induced subgraph of G if $S \subseteq V$ and $E(S) = \{(u,v) | (u,v) \in E, u \in S, v \in S\}$. According to [28], a k-core of a graph G is defined as follows.

Definition 2.1 (k-core). Given a graph G, the k-core of G, denoted by C_k, is a maximal subgraph of G in which every vertex has a degree at least k, i.e., $\forall v \in C_k,\ d_v(C_k) \geq k$.

Based on Definition 2.1, the core number of a vertex v, denoted by $\text{core}(v)$, is the largest integer k such that there is a k-core containing v. Denote by $k_{\max}$ the maximum k value such that a k-core of G exists, i.e., the maximum core number. It is easy to verify that the k-cores satisfy a containment property, i.e., $C_{k+1} \subset C_k$ for all $1 \leq k < k_{\max}$. The core decomposition of G is a problem of computing the core numbers for all vertices in G. Note that the core decomposition of a graph G can be computed in linear time by a classic peeling algorithm [6], which iteratively removes the minimum-degree node in G using an elegant bin-sort data structure.

Similar to the definition of k-core, Bonchi et al. [10] recently introduced a distance-generalized k-core notion, called (k,h)-core, based on the h-hop degrees of the vertices. Specifically, we denote by $dis_G(u,v)$ the shortest-path distance between u and v in G. Given a positive integer h, the h-hop neighborhood of a vertex v in G is defined as $N_v^h(G) \triangleq \{u | u \neq v, u \in V, dis_G(u,v) \leq h\}$. The h-hop degree of a vertex v in G, denoted by $d_v^h(G)$, is the cardinality of $N_v^h(G)$, i.e., $d_v^h(G) = |N_v^h(G)|$. If the context is clear, we use d_v^h and N_v^h to denote $d_v^h(G)$ and $N_v^h(G)$ respectively.

Definition 2.2 ((k,h)-core). Given a graph G and two integers k and h ($h > 0$), the (k,h)-core of G is a maximal subgraph C_k^h such that every vertex v in C_k^h has an h-hop degree at least k, i.e., $\forall v \in C_k^h,\ d_v^h(C_k^h) \geq k$.

It is worth noting that in Definition 2.2, the h-hop degree for each vertex in C_k^h is defined on the subgraph C_k^h (not on the original graph G). When $h = 1$, we can easily show that the (k,h)-core is the same as the traditional k-core.

As shown in [10], the (k,h)-core of a graph G is unique for any positive integer h. For a positive integer h, the (k,h)-core number of a vertex v, denoted by $\text{core}_h(v)$, is the largest integer k such that

there is a (k,h)-core containing v. Let $k^h_{\max}$ be the maximum k value such that a (k,h)-core of G exists, i.e., the maximum (k,h)-core number of G. Then, similar to the traditional k-cores, the (k,h)-cores of G also satisfy a containment property, i.e., $C^h_{k+1} \subseteq C^h_k$ for all $1 \leq k < k^h_{\max}$.

For a positive integer h, the distance-generalized core decomposition of G is a problem of determining the (k,h)-core numbers for all vertices in G. Below, we formally define our problem.

Problem statement. Given a graph G and a positive integer h, our goal is to compute the (k,h)-core number for each vertex in G.

3 EXISTING SOLUTIONS

In this section, we introduce several existing solutions proposed in [10] to compute the (k,h)-core decomposition. Similar to the traditional core decomposition algorithm, the (k,h)-core decomposition algorithm proposed in [10] is also based on a *peeling* idea. In particular, the peeling algorithm iteratively removes the vertex with the smallest h-hop degree and sets the (k,h)-core number as its h-hop degree at the time of removal. The detailed procedure of the peeling algorithm is shown in Algorithm 1.

The algorithm first computes the h-hop degree for each vertex $v \in V$ (line 3), and uses a bucketing array B to maintain all the vertices in V that have the same h-hop degree (line 4). Then, the algorithm iteratively deletes the vertices in V based on the non-decreasing order of the h-hop degrees of the vertices (lines 5-12). Specifically, in k-th iteration, the algorithm sequentially removes each vertex v in $B[k]$ (the h-hop degrees of v is equal to k) and sets its (k,h)-core numbers as k (lines 6-8). After that, the algorithm updates the h-hop degrees of the vertices in v's h-hop neighborhood (N^h_v), because the h-hop degrees of the vertices in N^h_v may need to update after removing v. For each $u \in N^h_v$, the algorithm first recomputes the h-hop degree of u in the reduced subgraph $G(V \setminus \{v\})$ (line 10), and then moves u into $B[\max\{k, d^h_u(G(V \setminus \{v\}))\}]$ if necessary. It is easy to see that the number of iterations of the algorithm is at most n, as the h-hop degrees of the vertices in G are bounded by n. The time complexity of Algorithm 1 is $O(n\tilde{n}(\tilde{n}+\tilde{m}))$ [10], where $\tilde{n}$ and $\tilde{m}$ are the number of vertices and edges of the largest subgraph induced by the h-hop neighborhood of a vertex in V, respectively.

As analyzed in [10], the most time-consuming step in Algorithm 1 is to recompute the h-hop degrees of all the vertices in N^h_v when deleting a vertex v. To speed up the algorithm, Bonchi et al. [10] proposed two improved algorithms based on lower and upper bounding techniques, called h-LB and h-LB+UB respectively. In particular, the h-LB algorithm first estimates the lower bound of the (k,h)-core number for each vertex. Then, based on the lower bounds, the h-LB algorithm can avoid a number of useless h-hop degree re-computations for the vertices whose lower bounds are no less than the h-hop degree of the current removed vertex [10]. The h-LB+UB algorithm also leverages an upper bound of the (k,h)-core number for each vertex to further improve the efficiency. Specifically, the algorithm first applies the upper bounds of vertices to partition the graph into several nested subgraphs. Then, the algorithm invokes h-LB to compute (k,h)-cores in the induced subgraph $G(V[i])$ following a top-down manner, where $V[i]$ denotes a set of vertices with upper bounds no less than i. As shown in [10], the h-LB+UB algorithm is the state-of-the-art algorithm for computing the (k,h)-core decomposition.

Algorithm 1: The basic peeling algorithm [10]

```
Input: a graph G = (V, E) and a positive integer h
Output: core_h(v) for all v ∈ V
1  Initialize B[v] ← ∅ for each v ∈ V;
2  for v ∈ V do
3      Compute d^h_v;
4      B[d^h_v] ← B[d^h_v] ∪ {v};
5  for k = 1 to n do
6      while B[k] ≠ ∅ do
7          Pick and remove a vertex v from B[k];
8          core_h(v) ← k;
9          for u ∈ N^h_v do
10             Compute d^h_u(G(V \ {v}));
11             Move u to B[max{k, d^h_u(G(V \ {v}))}];
12         V ← V \ {v};
13 return core_h(v) for all v ∈ V;
```

Limitations of the existing solutions. Although the h-LB+UB algorithm is more efficient than the basic peeling algorithm, it is still very costly for handling medium-sized graphs given that $h \geq 3$. For example, as reported in [10], the h-LB+UB algorithm takes nearly one hour to compute the (k,h)-core decomposition on the social network Douban (154,908 vertices and 327,162 edges) when $h = 4$. The main defect of the h-LB+UB algorithm is that the algorithm still needs to frequently recompute the h-hop degrees of the vertices when peeling a vertex. For a relatively large h value (e.g., $h \geq 3$), the time overheads for recomputing h-hop degrees can be very high on large graphs. To circumvent this issue, in the following sections, we will propose several efficient algorithms which can dynamically update the h-hop degrees of the vertices when peeling a vertex, instead of recomputing the h-hop degrees. Due to the efficient h-hop degree updating technique, the proposed algorithms are much faster than the state-of-the-art h-LB+UB algorithm as confirmed in our experiments.

4 THE PROPOSED ALGORITHMS

In this section, we propose several efficient (k,h)-core decomposition algorithms based on a novel h-hop degree updating technique. Below, we first introduce the basic version of our (k,h)-core decomposition algorithm. Then, we will develop a bitmap technique to improve the time and space overheads of our basic algorithm. Finally, we will propose a more efficient sampling-based algorithm, as well as a parallelization technique to further improve the efficiency and scalability of the (k,h)-core decomposition algorithms.

4.1 The basic h-hop degree updating algorithm

Recall that the most time-consuming step in Algorithm 1 is to recompute the h-hop degrees of the vertices in N^h_v after peeling v (lines 9-10 of Algorithm 1). To alleviate the computational costs, we propose a novel h-hop degree updating technique based on the following key observations.

Note that when deleting v, only the vertices in N^h_v may need to update their h-hop degrees. For any vertex $u \notin N^h_v$, its h-hop degree keeps unchanged after removing v. For a vertex $u \in N^h_v$, the

question is how can we efficiently update the h-hop degree of u after deleting v, without recomputing its h-hop degree on $G(V \setminus \{v\})$ (i.e., $d_u^h(G(V \setminus \{v\}))$). Clearly, after deleting v, the h-hop degree of u may reduce by more than 1 if $h > 1$. In order to derive the exact gap between d_u^h and $d_u^h(G(V \setminus \{v\}))$, it is sufficient to consider the vertices in $N_v^{h-s} \cup \{v\}$, where $s = dis_G(u, v)$ is the shortest-path distance between u and v in G ($s \leq h$). Below, we give two key observations.

Observation 1. *Given a positive integer $h \in \mathbb{N}^+$ and a vertex $u \in N_v^h$, we have $S_u = N_u^h \setminus (N_v^{h-s} \cup \{v\}) \subseteq N_u^h(G(V \setminus \{v\}))$ for $s \leq h$, where S_u is the set of vertices that are still the h-hop neighborhoods of u after removing v from G.*

Proof. Clearly, for any vertex $w \in S_u$, we have $dis_G(w, v) > h - s$ by definition. To prove the observation, we consider two disjoint subsets of S_u: $A = \{w | w \in S_u, dis_G(w, v) > h\}$ and $B = \{w | w \in S_u, h - s < dis_G(w, v) \leq h\}$. First, we claim that for any vertex $w \in A$, we have $w \in N_u^h(G(V \setminus \{v\}))$. Since $w \in S_u \subset N_u^h$, we have $dis_G(w, u) \leq h < dis_G(w, v)$. That is to say, there does not exist any shortest path between u and w that passes through v. Therefore, after deleting v from G, the shortest-path distance between w and u does not affect, indicating that $dis_{G\setminus\{v\}}(w, u) \leq h$. Second, for any vertex $w \in B$, we have $dis_G(u, w) < dis_G(u, v) + dis_G(v, w)$. This is because $dis_G(u, w) \leq h$, $dis_G(u, v) = s$ and $dis_G(v, w) > h - s$. Therefore, any shortest-path between u and w does not pass through v, which suggests that $dis_{G(V\setminus\{v\})}(w, u) \leq h$. □

Based on the Observation 1, we can see that only the vertices in $N_v^{h-s} \cup \{v\}$ may affect the h-hop degree of u after deleting v for any $u \in N_v^h$. Below, we show that any vertex w in $N_v^{h-s} \cup \{v\}$ that satisfies $dis_{G(V\setminus\{v\})}(u, w) > h$ must be excluded in $N_u^h(G(V \setminus \{v\}))$.

Observation 2. *Given a positive integer $h \in \mathbb{N}^+$ and a vertex $u \in N_v^h$, we define $F_u \triangleq \{w | w \in N_v^{h-s}, dis_{G(V\setminus\{v\})}(u, w) > h\}$. Then, we have $N_u^h \setminus N_u^h(G(V \setminus \{v\})) = \{v\} \cup F_u$.*

Proof. Clearly, the vertex v is contained in $N_u^h \setminus N_u^h(G(V \setminus \{v\}))$. On the one hand, for any vertex $w \neq v$ and $w \in N_u^h \setminus N_u^h(G(V \setminus \{v\}))$, we have $dis_G(u, w) \leq h$ and $dis_{G(V\setminus\{v\})}(u, w) > h$. Therefore, the shortest path from u to w in G must past through v. Since $dis_G(u, v) = s$, we have $dis_G(v, w) \leq h - s$. In other words, $w \in N_v^{h-s}$ which indicates that $w \in F_u$ holds. On the other hand, for any vertex $w \neq v$ and $w \in F_u$, $w \notin N_u^h(G(V \setminus \{v\}))$ clearly holds (by the definition of F_u). Since $w \in N_v^{h-s}$ and $dis_G(u, v) = s$, we have $dis_G(u, w) \leq h$ by triangle inequality. Hence, we obtain that $w \in N_u^h$. This completes the proof. □

Based on the Observation 2, we can obtain that $d_u^h - d_u^h(G(V \setminus \{v\})) = 1 + |F_u|$. As a result, the key to update the h-hop degree of a vertex u after removing v is to identify the set F_u. Since the set N_v^{h-s} can be easily derived by N_v^h, the challenge is how can we efficiently compute $dis_{G(V\setminus\{v\})}(u, w)$ on the graph after removing v. Below, we prove an interesting result which indicates that the shortest-path distance $dis_{G(V\setminus\{v\})}(u, w)$ can be computed on the subgraph induced by N_v^h if $dis_{G(V\setminus\{v\})}(u, w) \leq h$.

Theorem 4.1. *Given a positive integer $h \in \mathbb{N}^+$, all shortest-paths between $u \in N_v^h$ and $w \in N_v^{h-s}$ on $G(V \setminus \{v\})$, which satisfy $dis_{G(V\setminus\{v\})}(u, w) \leq h$, are contained in the induced subgraph $G(N_v^h)$, where $s = dis_G(u, v)$. In other words, for any shortest path $P = (u, ..., w_i, ..., w)$ between u and w on $G(V \setminus \{v\})$, we have $w_i \in N_v^h$ for all $w_i \in P$.*

Proof. Let $G' = G(V \setminus \{v\})$. Suppose, to the contrary, that there exists a shortest-path $P = (u, ..., w', ..., w)$ between $u \in N_v^h$ and $w \in N_v^{h-s}$ on G' that satisfies $w' \notin N_v^h$. By this assumption, we have $dis_{G'}(u, w) = dis_{G'}(u, w') + dis_{G'}(w', w)$. Then, $dis_G(v, w') - dis_G(v, u) \leq dis_G(u, w') \leq dis_{G'}(u, w')$ holds by triangle inequality. Since $w' \notin N_v^h$ (by assumption), we have $dis_G(v, w') > h$. Thus, we have $h - s < dis_{G'}(u, w')$. Similarly, we have $dis_G(v, w') - dis_G(v, w) \leq dis_G(w', w) \leq dis_{G'}(w', w)$. Therefore, we get that $s = h - (h - s) < dis_{G'}(w', w)$. Putting it all together, we can derive that $h < dis_{G'}(u, w)$ which is a contradiction. □

Let $\bar{F}_u \triangleq \{w | w \in N_v^{h-s}, dis_{G(V\setminus\{v\})}(u, w) \leq h\} = N_v^{h-s} \setminus F_u$. By Theorem 4.1, $\bar{F}_u$ can be determined on the subgraph induced by N_v^h. As a result, we are also able to compute $|F_u|$ on the induced subgraph $G(N_v^h)$ (not on the entire graph $G(V \setminus \{v\})$). In other words, we only need to explore a small subgraph $G(N_v^h)$ to maintain the h-hop degrees for all vertices in N_v^h after removing v, without recomputing the h-hop degree for every vertex in N_v^h.

Based on such an efficient h-hop degree updating technique, we propose a new (k, h)-core decomposition algorithm, called KHCore, which is shown in Algorithm 2. Algorithm 2 is also a peeling algorithm which iteratively deletes the vertices with the minimum h-hop degree (lines 3-13 in Algorithm 2). The algorithm terminates when all vertices are deleted. However, unlike Algorithm 1, Algorithm 2 invokes a UpdateHNbr procedure (Algorithm 3) to update the h-hop degree for each vertex in N_v^h after removing v based on the results shown in Theorem 4.1 (line 9). Below, we describe the detailed implementation of Algorithm 3.

In Algorithm 3, we develop a new data structure, named Reach, to maintain the set of vertices that are reachable from $u \in N_v^h$ within h hops in the induced subgraph $G(N_v^h)$. Initially, for each $u \in N_v^h$, if $dis_G(v, u) < h$, Reach$(u) = \{u\}$, and otherwise Reach$(u) = \emptyset$ (lines 2-5). This is because when $dis_G(v, u) = h$, the h-hop degree

Algorithm 2: KHCore

Input: a graph $G = (V, E)$ and a positive integer h
Output: core$_h(v)$ for all $v \in V$

1 **for** $v \in V$ **do**
2 Compute d_v^h;
3 **while** $V \neq \emptyset$ **do**
4 $k \leftarrow \arg\min_{v \in V}\{d_v^h\}$;
5 $B \leftarrow \{v | v \in V, d_v^h = k\}$;
6 **while** $B \neq \emptyset$ **do**
7 Pick and remove a vertex v from B;
8 core$_h(v) \leftarrow k$;
9 $d^h(G(V \setminus \{v\})) \leftarrow$ UpdateHNbr(G, h, v);
10 **for** $u \in N_v^h$ **do**
11 **if** $d_u^h(G(V \setminus \{v\})) \leq k$ and $u \notin B$ **then**
12 $B \leftarrow B \cup \{u\}$;
13 $V \leftarrow V \setminus \{v\}$;
14 **return** core$_h(v)$ for all $v \in V$;

Algorithm 3: UpdateHNbr (G, h, v)

1 $G(R) = (R, E(R)) \leftarrow$ the subgraph induced by N_v^h;
2 **for** $u \in R$ **do**
3 **if** $dis_G(v, u) < h$ **then**
4 $\text{Reach}[0][u] \leftarrow \{u\}$; $\text{Reach}[1][u] \leftarrow \{u\}$;
5 **else** $\text{Reach}[0][u] \leftarrow \emptyset$; $\text{Reach}[1][u] \leftarrow \emptyset$;
6 $p \leftarrow 1$; $q \leftarrow 0$;
7 **for** $hop = 1$ *to* h **do**
8 $q \leftarrow p$; $p \leftarrow 1 - p$;
9 **for** $(u, w) \in E(R)$ **do**
10 $\text{Reach}[q][u] \leftarrow \text{Reach}[q][u] \cup \text{Reach}[p][w]$;
11 $\text{Reach}[q][w] \leftarrow \text{Reach}[q][w] \cup \text{Reach}[p][u]$;
12 **for** $u \in R$ **do**
13 $s \leftarrow dis_G(u, v)$; $d_u^h(G(V \setminus \{v\})) \leftarrow d_u^h - 1$;
14 **for** $w \in R$ *s.t.* $dis_G(v, w) \leq h - s$ **do**
15 **if** $w \notin \text{Reach}[q][u]$ **then**
16 $d_u^h(G(V \setminus \{v\})) \leftarrow d_u^h(G(V \setminus \{v\})) - 1$;
17 **return** $d_u^h(G(V \setminus \{v\}))$ for each vertex $u \in R$;

of u decreases by 1 after deleting v, and thus we do not need to maintain the Reach structure for u in this case (i.e., $\text{Reach}(u) = \emptyset$). Then, we can make use of a dynamic programming (DP) procedure to identify all the vertices in N_v^h that are reachable from u within h hops (lines 6-11). In particular, the DP procedure is based on the following results. Let R_u^s be the set of vertices that are reachable from u within s hops. Then, R_u^{s+1} can be obtained by merging the sets R_w^s for all $w \in N_u \cup \{u\}$, i.e., $R_u^{s+1} = \bigcup_{w \in N_u \cup \{u\}} R_w^s$. We can adopt the Reach structure to implement such a DP procedure which is shown in lines 6-11 of Algorithm 3. Subsequently, Algorithm 3 applies the results in Theorem 4.1 to update the h-hop degree for each $u \in N_v^h$ (lines 12-16).

Complexity analysis. First, Algorithm 3 takes $O(d_v^h)$ time to initialize the Reach structures. Then, the algorithm takes $O(h|E(R)|d_v^h)$ time to compute the Reach sets (lines 7-12). This is because the size of the Reach set is bounded by d_v^h, and thus the set union operator can be computed in $O(d_v^h)$ time using some hash techniques. Finally, the time cost for updating the h-hop degrees in line 13-17 is $O(d_v^h \times d_v^{h-1})$. Let $\tilde{n}$ and $\tilde{m}$ be the number of vertices and edges of the largest subgraph induced by the h-hop neighborhood of a vertex in V, respectively. Then, the worst-case time complexity of Algorithm 3 is bounded by $O(\tilde{n}^2 + h\tilde{n}\tilde{m})$. Based on this, we can easily derive that the worst-case time complexity of Algorithm 2 is $O(n\tilde{n}^2 + nh\tilde{n}\tilde{m})$, which is asymptotically the same as the time complexity of Algorithm 1 (because h is often a very small integer). For the space overhead, we need to maintain the Reach sets for all vertices in N_v^h when deleting a vertex v which takes at most $O((d_v^h)^2) \leq O(\tilde{n}^2)$ in total. Therefore, the space complexity of Algorithm 2 can be bounded by $O(m + n + \tilde{n}^2)$. Below, we propose a bitmap technique to further improve the time and space overheads of our algorithm.

4.2 A bitmap optimization

Recall that in Algorithm 3, we have a Reach structure for each vertex $u \in N_v^h$ which maintains the set of vertices in N_v^h that are reachable from u within h hops. To improve the efficiency of the algorithm,

Algorithm 4: BmUpdateHNBr (G, h, v)

1 $G(R) = (R, E(R)) \leftarrow$ the subgraph induced by $R = N_v^h$;
2 Initialize the bitmaps (the Reach arrays) for all $u_i \in R$ to 0;
3 $N_v^{h-1} \leftarrow \{u_i \in R | dis_G(v, u_i) < h\}$; $d \leftarrow |N_v^{h-1}|$;
4 **for** $u_i \in N_v^{h-1}$ **do**
5 $\text{Reach}[0][i][\text{div}(i, 64)] \leftarrow 1 \ll \text{mod}(i, 64)$;
6 $\text{Reach}[1][i][\text{div}(i, 64)] \leftarrow 1 \ll \text{mod}(i, 64)$;
7 $p \leftarrow 1$; $q \leftarrow 0$;
8 **for** $hop = 1$ *to* h **do**
9 $q \leftarrow p$; $p \leftarrow 1 - p$;
10 **for** $(u_i, u_j) \in E(R)$ **do**
11 **for** $b = 0$ *to* $\text{div}(d, 64)$ **do**
12 $\text{Reach}[q][i][b] \vee = \text{Reach}[p][j][b]$;
13 $\text{Reach}[q][j][b] \vee = \text{Reach}[p][i][b]$;
14 **for** $u_i \in R$ **do**
15 $s \leftarrow dis_G(u_i, v)$; $d_{u_i}^h(G(V \setminus \{v\})) \leftarrow d_{u_i}^h - 1$;
16 **for** $u_j \in R$ *s.t.* $dis_G(v, u_j) \leq h - s$ **do**
17 **if** $((1 \ll \text{mod}(j, 64)) \wedge \text{Reach}[q][i][\text{div}(j, 64)]) = 0$
 then $d_{u_i}^h(G(V \setminus \{v\})) \leftarrow d_{u_i}^h(G(V \setminus \{v\})) - 1$;
18 **return** $d_{u_i}^h(G(V \setminus \{v\}))$ for each vertex $u_i \in R$;

we develop a bitmap to implement such a Reach structure for each vertex $u \in N_v^h$. Suppose without loss of generality that the vertices in N_v^h are labeled from u_0 to $u_{d_v^h - 1}$. For each vertex $u_i \in N_v^h$, we create a bitmap to represent the Reach structure of u_i. If u_j ($j \neq i$, $j \in \{0, 1, \cdots, d_v^h - 1\}$) is reachable within h hops from u_i in the subgraph induced by N_v^h, the j-th bit of u_i's bitmap is equal to 1, and otherwise it equals 0. For example, if u_i's bitmap is 10101, we can conclude that u_i can reach u_0, u_2, and u_4 within h hops in the induced graph $G(N_v^h)$. To merge two Reach sets, we can perform a *bitwise-or* operator using two bitmaps which is much more efficient than the traditional set-union operator. In this sense, the bitmap technique is not only reduce the space usage, but it also improves the time overhead of our algorithm.

Implementation details. The detailed implementation of the bitmap technique is outlined in Algorithm 4. Specifically, we make use of a set of 64-bit integers to represent a bitmap $\text{Reach}(u_i)$ for each vertex $u_i \in N_v^h$. In other words, the bitmap of a vertex u_i (i.e., $\text{Reach}(u_i)$) is an integer array. For any vertex u_i, if u_j is reachable from u_i within h hops in $G(N_v^h)$, then we can compute the position of u_j in u_i's bitmap array by $\text{div}(j, 64) = \left\lfloor \frac{j}{64} \right\rfloor$. In Algorithm 4, for each vertex $u_i \in N_v^h$, we first initialize its bitmap to 0 (line 1 of Algorithm 4). Then, for each vertex u_i, we set the i-th bit of u_i's bitmap to 1 (lines 4-6), denoting that the Reach set of u_i contains u_i itself. Note that in Algorithm 4, the notation $\text{mod}(i, 64)$ means $i\%64$ (lines 5-6), which is used to determine the bit-position of u_i in a bitmap. After that, we perform the DP procedure to compute the Reach sets. Note that the process of merging two Reach sets is implemented by a *bitwise-or* operator (lines 11-13). Finally, Algorithm 4 updates the h-hop degrees for all vertices in N_v^h (lines 14-17). Notice that based on the bitmap structure, we can use a *bitwise-and* operator to determine whether a vertex $u_j \in N_v^{h-s}$ is reachable from u_i within h hops (line 17).

Algorithm 5: KHCoreSamp

Input: a graph $G = (V, E)$, a positive integer h, and a sampling rate r
Output: $\text{core}_h(v)$ for all $v \in V$

1 Lines 1-2 of Algorithm 2;
2 $S \leftarrow$ uniformly sampling $r|V|$ vertices from V;
3 $\text{select}[v] \leftarrow |\{u | u \in N_v^h, u \in S\}|$ for each $v \in V$;
4 $\text{rate}[v] \leftarrow \text{select}[v]/d_v^h$ for each $v \in V$;
5 Lines 3-8 of Algorithm 2;
6 $d^h(G(V \setminus \{v\})) \leftarrow$ UpdateHNbrSamp(G, h, v, S, select, rate);
7 Lines 10-14 of Algorithm 2;

Complexity analysis. Armed with the bitmap technique, Algorithm 4 can significantly reduce the set-union costs. In our basic KHCore algorithm (Algorithm 3), the set-union operator can be done in $O(d_v^h)$ time (lines 10-12 of Algorithm 3). However, by using the bitmap technique, we can implement the set union operator by a *bitwise-or* operator which takes $O(d_v^h/64)$ time. In other words, the bitmap technique can achieve around 64× speedup for the set union computation. As a result, the total time costs of the KHCore algorithm with bitmap technique can be bounded by $O(n\tilde{n}^2 + nh\tilde{n}\tilde{m}/64)$. Since h is typically smaller than 64, the time complexity of our algorithm is lower than that of Algorithm 1 which is confirmed in our experiments.

Remark. It is worth remarking that the lower and upper bounding techniques developed in [10] can also be integrated into Algorithm 2. However, we empirically find that such lower and upper bounding techniques cannot significantly improve the efficiency of our algorithm, thus in this work we mainly focus on our algorithms without using the lower and upper bounds developed in [10]. Also, it is worth emphasizing that the bitmap technique is an elegant implementation of our theoretical finding; it is not a general optimization technique and it cannot be used in the state-of-the-art algorithm [10]. In the experiments, we will focus mainly on evaluating the proposed algorithms with the bitmap implementation.

4.3 A sampling-based algorithm

To further improve the efficiency, we propose a sampling-based algorithm to compute the (k, h)-core decomposition. The key idea of the sampling-based algorithm is that when deleting a vertex v, it estimates the updated h-hop degree for a vertex $u \in N_v^h$ using the randomly sampled vertices (not all vertices in N_v^h). Due to the less computation for updating the h-hop degrees of vertices, the sampling-based approach can significantly reduce the time cost compared to the exact algorithm.

The implementation details of the sampling-based algorithm are shown in Algorithm 5. First, the algorithm randomly selects $r|V|$ vertices from V (line 2 of Algorithm 5), where $0 < r < 1$ denotes the sampling rate. Then, for each vertex v, the algorithm computes the number of selected vertices in the h-hop neighborhood of v (line 3), denoted by $\text{select}[v]$. Based on $\text{select}[v]$, the algorithm calculates the sampling rate for v (line 4 of Algorithm 5), i.e., $\text{rate}[v] = \text{select}[v]/d_v^h$. Similar to Algorithm 2, the algorithm iteratively deletes the vertex that has the smallest h-hop degree (lines 5-7). When removing a vertex v, it invokes Algorithm 6 to update the h-hop degrees of the vertices in N_v^h (line 6).

Algorithm 6: UpdateHNbrSamp (G, h, v, S, select, rate)

1 Lines 1-2 of Algorithm 4;
2 $\tilde{N}_v^{h-1} \leftarrow N_v^{h-1} \cap S$; $R = N_v^h$; $d \leftarrow |\tilde{N}_v^{h-1}|$;
3 Lines 4-13 of Algorithm 4;
4 **for** $u_i \in R$ **do**
5 $s \leftarrow dis_G(u_i, v)$; cnt $\leftarrow 0$;
6 **if** $v \in S$ **then** cnt $\leftarrow 1$;
7 **for** $u_j \in R \cap S$ *s.t.* $dis_G(v, u_j) < h - s$ **do**
8 **if** $(1 \ll \text{mod}(j, 64)) \wedge \text{Reach}[q][i][\text{div}(j, 64)] = 0$ **then**
9 cnt $\leftarrow$ cnt + 1;
10 $\text{select}[u_i] \leftarrow \text{select}[u_i] - \text{cnt}$;
11 $d_{u_i}^h(G(V \setminus \{v\})) \leftarrow \text{select}[u_i]/\text{rate}[u_i]$;
12 **return** $d_{u_i}^h(G(V \setminus \{v\}))$ for each vertex $u_i \in N_v^h$;

In Algorithm 6, it first initializes the bitmap structures for the vertices in N_v^h (lines 1-2 of Algorithm 6). Let S be the set of sampled vertices. Then, the algorithm computes the bitmaps for the vertices in $N_v^{h-1} \cap S$ (lines 2-3). Note that for the vertices in $N_v^h \setminus N_v^{h-1}$, their h-hop degrees decrease by 1 after deleting v, thus we do not need to maintain the bitmaps for those vertices. Subsequently, for each $u_i \in N_v^h$, the algorithm updates the h-hop degree of u_i based on the sampled vertices (lines 4-11). Notice that it first updates $\text{select}[u_i]$, and then uses $\text{select}[u_i]/\text{rate}[u_i]$ as an estimator for the updated $d_{u_i}^h$ (lines 10-11).

Complexity analysis. We first analyze the time complexity of Algorithm 6. Compared to Algorithm 4, Algorithm 6 only need to maintain the bitmaps for the sampled vertices $N_v^{h-1} \cap S$. The cardinality of the set $N_v^{h-1} \cap S$ can be bounded by $O(rd_v^h) \leq O(r\tilde{n})$. Similar to Algorithm 4, we can easily derive that the time complexity of Algorithm 6 is $O(r\tilde{n}^2 + hr\tilde{n}\tilde{m}/64)$, where $r < 1$ is sampling rate. Based on this, the time complexity of Algorithm 5 is $O(rn\tilde{n}^2 + hrn\tilde{n}\tilde{m}/64)$, which is lower than our exact algorithm by a factor r. For example, if $r = 0.1$, the sampling-based algorithm can be one order of magnitude faster than the proposed exact algorithm, as confirmed in our experiment. For the space usage, we can easily derive that the complexity of the sampling-based algorithm is the same as that of the exact algorithm.

4.4 Parallelization

In this section, we explore how Algorithm 2 splits the computation in several sub-tasks which can be processed independently. Note that the parallelization strategy for Algorithm 2 and Algorithm 5 is the same. Therefore, we focus mainly on developing parallelization strategy for Algorithm 2.

First, in lines 1-2 of Algorithm 2, we can compute the h-hop degree for each vertex in parallel, because the sub-tasks for computing h-hop degrees are clearly independent. Second, when deleting the vertices in the bucket B (line 6 of Algorithm 2), we can also process the vertices in parallel. However, the sub-task for deleting a vertex is not independent, but it depends on the former deleted vertices. To make all the sub-tasks independent, we can follow an increasing order by vertex ID to delete vertex. When processing a vertex v_i, we use a thread to update the h-hop degrees of the vertices in $N_{v_i}^h$ that either has a h-hop degree no less than $d_{v_i}^h$ or has a larger vertex ID.

Based on this strategy, the sub-tasks for removing the vertices in the bucket B are independent, and therefore we can safely process the vertices in B in parallel. Note that in Algorithm 4, the procedure of updating the h-hop degree of a vertex should be considered as an atomic operator (line 15 and line 18). In our experiments, we will show that the proposed parallel algorithms can achieve a very good speedup ratio over the corresponding sequential algorithms.

5 EXPERIMENTS

In this section, we conduct extensive experiments to evaluate the efficiency and scalability of the proposed algorithms. Below, we first describe the experimental setup and then report our results.

5.1 Experimental setup

We implement three sequential algorithms to compute the (k, h)-core decomposition: KHC, KHCS, and h-LB+UB. The KHC and KHCS are our exact and sampling-based (k, h)-core decomposition algorithms respectively. Both KHC and KHCS are integrated with the bitmap technique proposed in Section 4.2. The h-LB+UB algorithm denotes the state-of-the-art h-LB+UB algorithm [10], which is served as a baseline in our experiments. For all these algorithms, we also implement the parallelized versions using OpenMP. All algorithms are implemented in C++. We conduct all experiments on a PC with two 2.3 GHz Xeon CPUs (16 cores in total) and 64GB memory running Ubuntu 16.4.

Datasets. We make use of 12 real-world datasets in our experiments. Table 1 shows the detailed statistics of the datasets, where $d_{\max}$, Δ and $k_{\max}$ denote the maximum degree, the diameter and the maximum k-core number of the network. ca-AstroPH (ca-As for short) is a collaboration network; com-amazon (Amazon) is a co-purchasing network; Douban, Hyves, soc-LiveJournal (SocLJ), soc-youtube (Socytb), soc-pokec (Pokec), and soc-Epinions (SocEps) are social networks; flickrEdges (Flickr) is a network of Flickr images sharing common metadata such as tags, groups, locations etc; bio-CE-CX (BioCE) and bio-WormNet-v3 are biological networks; italycnr-2000 (Cnr2000) is a web graph. All datasets can be downloaded from http://networkrepository.com and http://snap.stanford.edu/data.

Parameters. Both KHC and h-LB+UB have only one parameter $h \in \mathbb{N}^+$, and the KHCS algorithm has an additional parameter r which denotes the sampling rate. In our experiment, the parameter h is selected from the interval $[2, 5]$ (the same parameter setting also used in [10]), because larger values are often not interesting in practice [10]. For KHCS, the parameter r is selected from the interval $[0.05, 0.8]$ with a default value of $r = 0.1$, because KHCS performs very well on all datasets given that $r = 0.1$.

5.2 Experimental results

Exp-1: Efficiency of various sequential algorithms. We start by comparing the efficiency of different sequential algorithms. Fig. 2 shows the runtime of h-LB+UB, KHC, and KHCS on all datasets. Note that in all experiments, INF means that the algorithm does not terminate in 28 hours. From Fig. 2(a), we observe that KHC and KHCS significantly outperform the state-of-the-art h-LB+UB algorithm on most datasets with $h = 2$. We also notice that on some very sparse graphs, such as Amazon and Hyves, h-LB+UB is faster than KHC and KHCS. This is because, on very sparse graphs, the costs for recomputing the h-hop degrees are very low with $h = 2$. However, when $h \geq 3$ (Figs. 2(b-d)), we can clearly see that KHC and KHCS are substantially faster than h-LB+UB on all datasets. For example, on BioCE, KHC is at least one order of magnitude faster than h-LB+UB with $h \geq 3$. On larger datasets, such as Pokec (more than 1.6 million vertices and 22 million edges), h-LB+UB cannot terminate within 28 hours when $h = 3$, while KHC takes around 52,000 seconds to compute all (k, h)-cores. When comparing KHC with KHCS, we find that KHCS (with the sampling rate $r = 0.1$) is much more efficient than KHC given that $h \geq 3$. On some large graphs, KHCS is one order of magnitude faster than KHC when $h \geq 3$. For instance, on Pokec, KHCS takes around 2,000 seconds to compute all (k, h)-cores when $h = 3$, whereas the time overhead of KHC is around 52,000 seconds. In addition, when $h = 5$ (Fig. 2(d)), h-LB+UB cannot handle four medium-sized graphs, while our algorithms still work well on all eight medium-sized graphs. These results are consistent with our theoretical analysis in Section 4.

Table 1: Datasets

Dataset	$\|V\|$	$\|E\|$	$d_{\max}$	Δ	$k_{\max}$
BioCE	15,229	245,952	375	13	78
BioWorm	16,347	762,822	1,272	12	164
ca-As	18,771	198,050	504	14	56
SocEps	75,880	405,740	3,044	15	67
Flickr	105,939	2,316,948	5,425	9	573
Douban	154,908	327,162	287	9	15
Cnr2000	325,557	2,738,969	18,236	34	83
Amazon	334,863	925,872	549	44	6
Socytb	495,957	1,936,748	25,409	21	49
Hyves	1,402,673	2,777,419	31,883	10	39
Pokec	1,632,803	22,301,964	14,854	14	47
SocLJ	4,846,609	42,851,237	20,333	16	372

Exp-2: Efficiency of different parallel algorithms. Here we evaluate the performance of the parallelized versions of h-LB+UB, KHC, and KHCS. To this end, we vary the number of threads t from 1 to 16 with different h values. Fig. 3 shows the results on the Flickr dataset, and similar results can also be observed on the other datasets. As expected, the runtime of all the three algorithms decreases with increasing t. We also observe that if $t \geq 8$, the speedup ratios of all algorithms do not significantly increase as t grows. This is because, for all algorithms, the parallel performance mainly relies on the size of the bucket B that maintains all the vertices having the minimum h-hop degrees. In some iterations of each algorithm, the size of the bucket B might be smaller than t which limits the parallel speedup ratio of the algorithm. In addition, we also notice that the speedup ratio of KHCS is significantly higher than those of h-LB+UB and KHC. For example, when $h = 3$, the parallel KHCS algorithm with $t = 16$ can achieve nearly 9× speedup over the sequential KHCS algorithm on Flickr (Fig. 3(b)). However, the speedup ratios of the parallel h-LB+UB and KHC algorithms are around 6.6 and 5.3 on Flickr respectively, given $t = 16$ and $h = 3$.

Exp-3: Runtime of KHCS with varying r. We evaluate the runtime of KHCS with varying r (sampling rate). Fig. 4 depicts the runtime of (parallel) KHCS when r varies from 0.05 to 0.8. As expected, the runtime of KHCS increases when r increases, because the graph is sparser with a smaller r value. In addition, we also

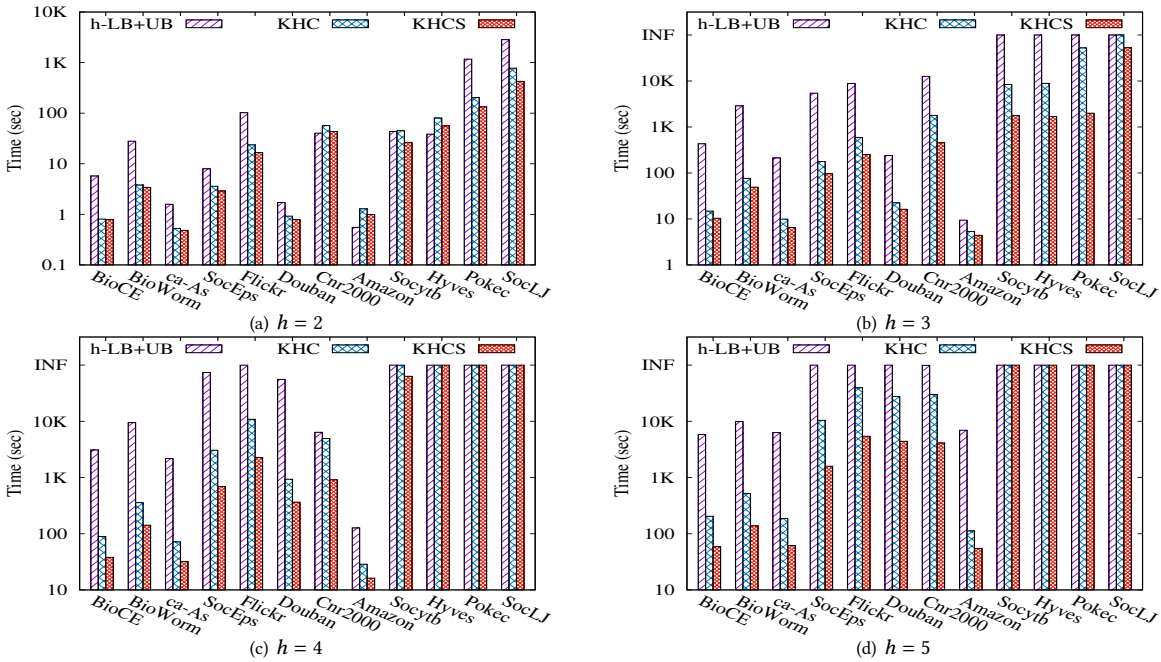

Figure 2: Runtime of different sequential algorithms on all datasets

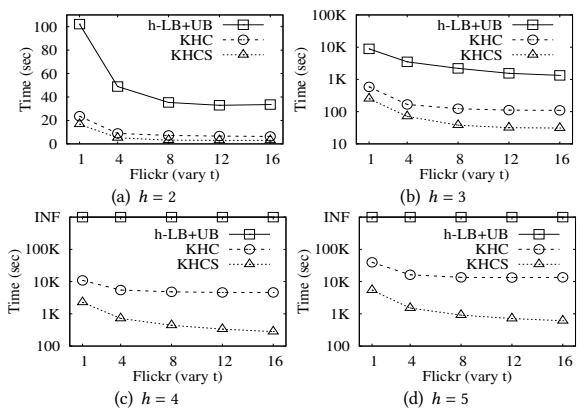

Figure 3: Runtime of different parallel algorithms

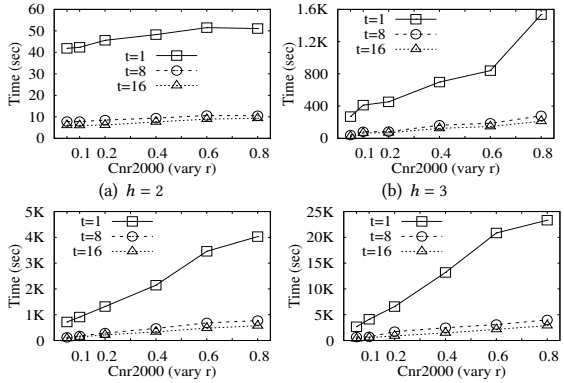

Figure 4: Runtime of the KHCS algorithm

observe that KHCS can always achieve high speedup ratios at different sampling rates. For example, when $h = 3$ and $r = 0.2$, KHCS takes 453 seconds to compute all (k, h)-cores using a single thread, while it only takes 83 seconds and 65 seconds using 8 and 16 threads, respectively. These results further confirm the high efficiency of our parallel KHCS algorithm.

Exp-4: Precisions of KHCS with varying r. In this experiment, we evaluate the precision of the KHCS algorithm with various sampling rates. Here we define the precision as follows. Let $\text{core}_h[v]$ and $\widehat{\text{core}}_h[v]$ be the exact and the estimated (k, h)-core number of the vertex v, respectively. Then, the precision of an algorithm is computed by $1 - (\sum_{v \in V}(|\text{core}_h[v] - \widehat{\text{core}}_h[v]|)/\text{core}_h[v])/|V|$. Fig. 5 shows the precisions of KHCS with varying r on five datasets. Similar results can also be observed on the other datasets. As expected, the precisions of KHCS typically increase as r increases. When $h = 2$ (Fig. 5(a)), the precisions of KHCS are no less than 92% on all datasets even when $r = 0.05$. Moreover, with r increases, the precisions can be quickly improved to 98% on all datasets given that $h = 2$. When $h \geq 3$ (Fig. 5(b-d)), KHCS exhibits very high precisions ($\geq$ 99%) in most cases. For example, even when $r = 0.05$, the precision of KHCS is higher than 99% with $h \geq 4$ on most datasets. These results indicate that KHCS is very accurate in practice even for a very small sampling rate (e.g., $r = 0.1$).

Exp-5: Memory overhead. We compare the memory overhead of different algorithms. Fig. 6 shows the results on Flickr and Cnr2000, and similar results can also be obtained on the other datasets. As expected, the memory overheads of KHC and KHCS are slightly higher than that of the h-LB+UB algorithm, because our algorithms need to maintain a Reach data structure (the bitmaps for all vertices). Specifically, we can see that the memory usage of h-LB+UB is less than twice of the graph size. The memory overhead of KHC and KHCS are comparable, both of which are less than 4 times of the graph size. These results indicate that our algorithms (with the bitmap optimization technique) are space efficient for handling real-world graphs.

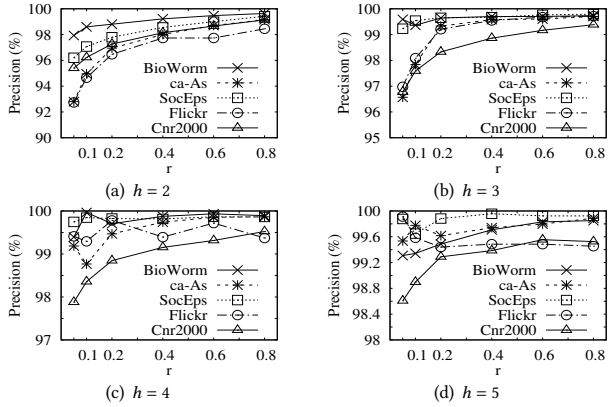

Figure 5: Precisions of KHCS with varying r

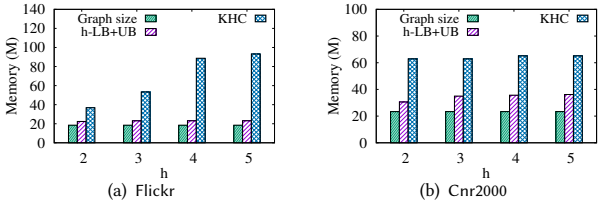

Figure 6: Memory overheads of various algorithms

Exp-6: Scalability. Here we aim at evaluating the scalability of h-LB+UB, KHC and KHCS, using 16 threads. To this end, we first generate eight subgraphs by randomly sampling 20-80% of vertices and edges from the original graph respectively. Then, we evaluate the runtime of all algorithms on these subgraphs using 16 threads. The results on Pokec with $h = 2$ and $h = 3$ are shown in Fig. 7, and the results on the other datasets and for the other h values are consistent. From Fig. 7, we observe that the time costs of KHC and KHCS increase smoothly as $|V|$ or $|E|$ increases. The runtime of h-LB+UB, however, increases sharply with increasing $|V|$ or $|E|$. Moreover, both KHC and KHCS significantly outperform h-LB+UB under all parameter settings. These results suggest that both KHC and KHCS exhibit a good scalability, while h-LB+UB shows a poor scalability when $h \geq 3$.

6 RELATED WORK

***K*-core based models and algorithms.** The k-core model was originally proposed by Seidman [28] for modeling cohesive subgraphs in an undirected network. Recently, many k-core based models have been proposed for modeling cohesive subgraphs on different types of networks. For example, Batagelj and Zaversnik [7] introduced a generalized concept of k-core by considering weights of the edges on weighted graphs. Bonchi et al. [9] proposed a k-core model for uncertain graphs based on a definition of *reliable* degree of nodes. Li et al. [22] proposed an influential community model based on k-core to capture both the influence and cohesiveness of a community. Galimberti et al. proposed two generalized k-core models for multi-layer networks [18] and temporal graphs [17], respectively. Fang et al. [16] extended the k-core concept to attribute graphs. More recently, Li et al. [21] proposed a skyline k-core model for modeling communities on multi-valued networks. From the algorithmic point of view, Batagelj and Zaversnik [6] proposed a linear-time core decomposition algorithm. Sariyüce et al. [25] and Li et al. [23] developed efficient algorithms for maintaining the core decomposition on dynamic graphs. Wen et al. [33] presented an I/O efficient core decomposition algorithm for web scale graphs. Unlike all these existing studies, we focus on developing efficient algorithms to solve the distance-generalized core decomposition problem, which was originally introduced in [10].

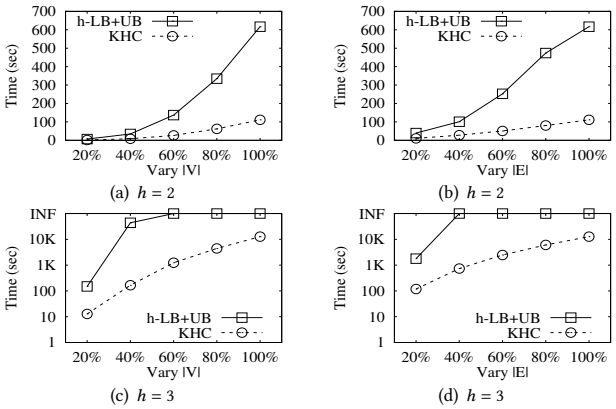

Figure 7: Scalability testing on the Pokec dataset (16 threads)

Other cohesive subgraph models. Beyond k-core, there also exist many other cohesive subgraph models which have been widely used for modeling communities. Notable examples include the maximal clique model [11, 12], the k-plex model [8, 29], the k-truss model [13, 20, 32], the nucleus model [26, 27], the locally densest subgraph (LDS) model [15, 24, 31], as well as the maximal k-edge connected subgraph (k-ECS) model [1, 35]. Noted that the problems of enumerating all maximal cliques and all k-plex subgraphs are NP-hard [8, 11], thus they are often intractable for massive graphs. However, for the k-truss, the nucleus, the LDS, the k-ECS models, there exist polynomial-time algorithms to compute the corresponding cohesive subgraphs. Similar to these cohesive subgraph models, the (k, h)-core model studied in the paper can also be computed in polynomial time [10].

7 CONCLUSION

In this paper, we propose an efficient peeling algorithm to compute the (k, h)-core decomposition on graphs based on a novel h-hop degree updating technique. The striking feature of our algorithm is that it only needs to traverse a small induced subgraph ($G(N_v^h)$) to maintain the h-hop degrees for all vertices after peeling a vertex v, instead of recomputing the h-hop degrees of the vertices. We also develop an elegant bitmap technique to efficiently implement such an h-hop degree updating procedure. Additionally, we present a sampling-based algorithm and a parallelization strategy to further improve the efficiency for (k, h)-core decomposition. The results of extensive experiments on 12 real-world large graphs demonstrate the efficiency and scalability of the proposed algorithms.

8 ACKNOWLEDGMENTS

This work was partially supported by (i) National Key Research and Development Program of China 2020AAA0108503, (ii) NSFC Grants 62072034 and 61772346, (iii) ARC FT200100787 and DP210101347, and (iv) CCF-Baidu Open Fund. Rong-Hua Li is the corresponding author of this paper.

REFERENCES

[1] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Linear-time enumeration of maximal K-edge-connected subgraphs in large networks by random contraction. In *CIKM*. 909–918.

[2] Md Altaf-Ul-Amine, Kensaku Nishikata, Toshihiro Korna, Teppei Miyasato, Yoko Shinbo, Md Arifuzzaman, Chieko Wada, Maki Maeda, Taku Oshima, Hirotada Mori, et al. 2003. Prediction of protein functions based on k-cores of protein-protein interaction networks and amino acid sequences. *Genome Informatics* 14 (2003), 498–499.

[3] J. Ignacio Alvarez-Hamelin, Luca Dall'Asta, Alain Barrat, and Alessandro Vespignani. 2005. Large scale networks fingerprinting and visualization using the k-core decomposition. In *NIPS*. 41–50.

[4] Gary D. Bader and Christopher W. V. Hogue. 2003. An automated method for finding molecular complexes in large protein interaction networks. *BMC Bioinformatics* 4 (2003), 2.

[5] Balabhaskar Balasundaram, Sergiy Butenko, and Illya V. Hicks. 2011. Clique Relaxations in Social Network Analysis: The Maximum k-Plex Problem. *Operations Research* 59, 1 (2011), 133–142.

[6] Vladimir Batagelj and Matjaz Zaversnik. 2003. An O(m) Algorithm for Cores Decomposition of Networks. *CoRR* cs.DS/0310049 (2003).

[7] Vladimir Batagelj and Matjaz Zaversnik. 2011. Fast algorithms for determining (generalized) core groups in social networks. *Adv. Data Analysis and Classification* 5, 2 (2011), 129–145.

[8] Devora Berlowitz, Sara Cohen, and Benny Kimelfeld. 2015. Efficient Enumeration of Maximal k-Plexes. In *SIGMOD*. 431–444.

[9] Francesco Bonchi, Francesco Gullo, Andreas Kaltenbrunner, and Yana Volkovich. 2014. Core decomposition of uncertain graphs. In *KDD*. 1316–1325.

[10] Francesco Bonchi, Arijit Khan, and Lorenzo Severini. 2019. Distance-generalized Core Decomposition. In *SIGMOD*. 1006–1023.

[11] Coenraad Bron and Joep Kerbosch. 1973. Finding All Cliques of an Undirected Graph (Algorithm 457). *Commun. ACM* 16, 9 (1973), 575–576.

[12] James Cheng, Yiping Ke, Ada Wai-Chee Fu, Jeffrey Xu Yu, and Linhong Zhu. 2011. Finding maximal cliques in massive networks. *ACM Trans. Database Syst.* 36, 4 (2011), 21:1–21:34.

[13] Jonathan Cohen. 2005. Trusses: Cohesive subgraphs for social network analysis. *Technical report, National Security Agency* (2005).

[14] Wanyun Cui, Yanghua Xiao, Haixun Wang, and Wei Wang. 2014. Local search of communities in large graphs. In *SIGMOD*. 991–1002.

[15] Maximilien Danisch, T.-H. Hubert Chan, and Mauro Sozio. 2017. Large Scale Density-friendly Graph Decomposition via Convex Programming. In *WWW*.

[16] Yixiang Fang, Reynold Cheng, Siqiang Luo, and Jiafeng Hu. 2016. Effective Community Search for Large Attributed Graphs. *PVLDB* 9, 12 (2016), 1233–1244.

[17] Edoardo Galimberti, Alain Barrat, Francesco Bonchi, Ciro Cattuto, and Francesco Gullo. 2018. Mining (maximal) Span-cores from Temporal Networks. In *CIKM*.

[18] Edoardo Galimberti, Francesco Bonchi, and Francesco Gullo. 2017. Core Decomposition and Densest Subgraph in Multilayer Networks. In *CIKM*. 1807–1816.

[19] Christos Giatsidis, Dimitrios M. Thilikos, and Michalis Vazirgiannis. 2011. Evaluating Cooperation in Communities with the k-Core Structure. In *ASONAM*. 87–93.

[20] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-truss community in large and dynamic graphs. *SIGMOD* (2014), 1311–1322.

[21] Rong-Hua Li, Lu Qin, Fanghua Ye, Jeffrey Xu Yu, Xiaokui Xiao, Nong Xiao, and Zibin Zheng. 2018. Skyline Community Search in Multi-valued Networks. In *SIGMOD*.

[22] Rong-Hua Li, Lu Qin, Jeffrey Xu Yu, and Rui Mao. 2015. Influential Community Search in Large Networks. *PVLDB* 8, 5 (2015), 509–520.

[23] Rong-Hua Li, Jeffrey Xu Yu, and Rui Mao. 2014. Efficient Core Maintenance in Large Dynamic Graphs. *IEEE Trans. Knowl. Data Eng.* 26, 10 (2014), 2453–2465.

[24] Lu Qin, Rong-Hua Li, Lijun Chang, and Chengqi Zhang. 2015. Locally Densest Subgraph Discovery. In *KDD*. 965–974.

[25] Ahmet Erdem Sariyüce, Bugra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V. Çatalyürek. 2013. Streaming Algorithms for k-core Decomposition. *PVLDB* 6, 6 (2013), 433–444.

[26] Ahmet Erdem Sariyüce, C. Seshadhri, Ali Pinar, and Ümit V. Çatalyürek. 2015. Finding the Hierarchy of Dense Subgraphs using Nucleus Decompositions. In *WWW*.

[27] Ahmet Erdem Sariyüce, C. Seshadhri, Ali Pinar, and Ümit V. Çatalyürek. 2017. Nucleus Decompositions for Identifying Hierarchy of Dense Subgraphs. *TWEB* 11, 3 (2017), 16:1–16:27.

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

[29] Stephen B. Seidman and Brian L. Foster. 1978. A graph-theoretic generalization of the clique concept. *Journal of Mathematical Sociology* 6, 1 (1978), 139–154.

[30] Carmi Shai, Havlin Shlomo, Kirkpatrick Scott, Shavitt Yuval, and Shir Eran. 2007. A model of Internet topology using k-shell decomposition. *PNAS* 104, 27 (2007), 11150–11154.

[31] Nikolaj Tatti and Aristides Gionis. 2015. Density-friendly Graph Decomposition. In *WWW*.

[32] Jia Wang and James Cheng. 2012. Truss Decomposition in Massive Networks. *PVLDB* 5, 9 (2012), 812–823.

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

[34] Yang Zhang and Srinivasan Parthasarathy. 2012. Extracting Analyzing and Visualizing Triangle K-Core Motifs within Networks. In *ICDE*. 1049–1060.

[35] Rui Zhou, Chengfei Liu, Jeffrey Xu Yu, Weifa Liang, Baichen Chen, and Jianxin Li. 2012. Finding maximal k-edge-connected subgraphs from a large graph. In *EDBT*. 480–491.