Hierarchical Core Decomposition in Parallel: From Construction to Subgraph Search

Deming Chu[§], Fan Zhang[⋆], Wenjie Zhang[§], Xuemin Lin[§], Ying Zhang[♯]

[§]*University of New South Wales,* [⋆]*Guangzhou University,* [♯]*University of Technology Sydney*
deming.chu@unsw.edu.au, fanzhang.cs@gmail.com
{zhangw,lxue}@cse.unsw.edu.au, ying.zhang@uts.edu.au

***Abstract*—The model of k-core discovers a novel hierarchical structure of a network, which has been widely applied in various areas, e.g., sociology, biology, and brain science. Based on the containment relations of k-cores with different k, the *hierarchical core decomposition* (HCD) of a graph formalizes the hierarchy of all k-cores for each possible k. HCD is effective in locating high-quality subgraphs (e.g., densest subgraph search) and exploring particular network phenomena (e.g., user engagement study). However, existing solutions of HCD are still not efficient enough, for both the hierarchy construction and the subgraph search on the hierarchy. In this paper, we propose the first parallel construction algorithm `PHCD` for HCD, using a new union-find-based paradigm, and the first parallel algorithm `PBKS` to search high-quality subgraphs from the hierarchy with respect to various community scoring metrics. We prove the problem of hierarchy construction is $\mathcal{P}$-complete (difficult to parallelize effectively). Despite the negative result, our `PHCD` has a near-linear time cost, and `PBKS` is time-optimal in score computation for most community metrics. Extensive experiments are conducted on 10 real-world networks, where our proposed parallel algorithms significantly outperform the existing solutions, for both the hierarchy construction and the subgraph search.**

I. Introduction

Graphs are ubiquitous to model complex relations between entities in the real world, such as social networks [1]–[3], web networks [4], [5], and biological networks [6]. As one of the most important graph concepts, cohesive subgraphs are formed by groups of densely connected vertices. The k-core is a well-studied cohesive subgraph model, which is defined as a maximal *connected* subgraph where every vertex is adjacent to at least k other vertices in the subgraph [7], [8]. Every vertex in the graph has a coreness value, i.e., the largest integer k such that the k-core contains the vertex.

The model of k-core can decompose a graph into an elegant hierarchy: for every integer k, each k-core is contained in exactly one $(k-1)$-core, and each k-core is disjoint from each other for the given k. Such a hierarchical core decomposition can be represented by a forest structure, where each tree node contains the vertices in a k-core with coreness of k, and each tree edge represents that a k-core contains another k'-core, i.e., the parent-child relation of their corresponding tree nodes.

Example 1 (Rationale of Hierarchical Core Decomposition)**.** *Figure 1(a): We depict a graph where the vertices are colored by their coreness values. The whole graph S_2 is a 2-core, in which the 4-core is S_4 and two 3-cores are represented by*

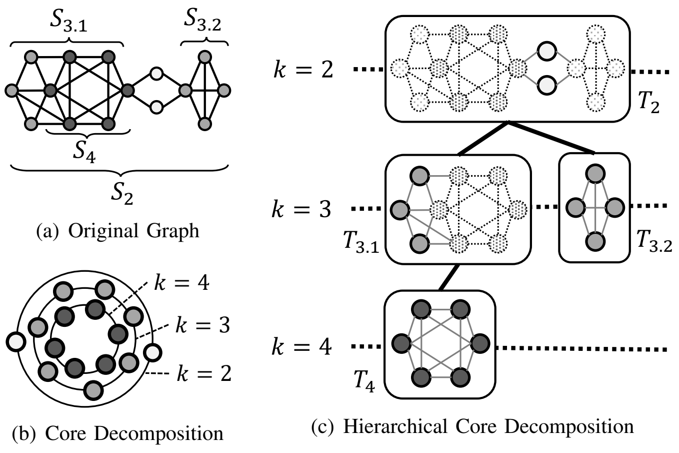


Fig. 1. Rationale of Hierarchical Core Decomposition

$S_{3.1}$ and $S_{3.2}$. Figure 1(b): The core decomposition organizes all vertices into a core-periphery structure, where the vertices with a higher coreness are more close to the center of the network. In a core decomposition, all vertices with the same coreness are maintained at the same layer. Figure 1(c): In the HCD, each tree node contains the vertices in a k-core with coreness k (marked by solid circles). Each k-core one-to-one corresponds to a tree node. Let T_i represent the tree node corresponding to S_i. The HCD (i) distinguishes the vertices with the same coreness while in different k-cores, e.g. the vertices in $T_{3.1}$ and $T_{3.2}$, and (ii) identifies all the k-cores and their containment relations, e.g., $S_{3.1} = G[S_4 + T_{3.1}]$ and $S_2 = G[S_{3.1} + S_{3.2} + T_2]$, where $G[S]$ is the subgraph induced by the vertices in S.

Example 2 (Subgraph Search on HCD)**.** *We can find the subgraphs with high quality using HCD, e.g., the k-core with the largest average degree (twice the number of edges divided by the number of vertices). In Figure 1, the 4-core S_4 has an average degree of 4, while the average degree of the 3-core $S_{3.1}$ is about 4.44. Because $S_{3.1}$ has the highest average degree among all k-cores for each possible k, we can return $S_{3.1}$ as the resulting subgraph.*

Applications. HCD is a fundamental concept in network analysis, which is effective to locate cohesive subgraphs and can lead to a better understanding of some network phenomena [9].

The representative applications of HCD are as follows.

(Cohesive Subgraph Search) We can efficiently search cohesive subgraphs related to k-core using HCD. For instance, the k-core with the highest average degree can be found on the HCD in $O(n)$ time [10], where n is the number of vertices in the graph. It is a 0.5-approximate solution for densest subgraph problem, which is the state-of-the-art on both output quality and time cost. HCD can also help other k-core related problems, e.g., finding the maximum clique [10], influential communities [11], and attributed communities [12].

(User Engagement Analysis) The coreness of a vertex (user) is often used to estimate the engagement level of the user [13]. It is validated that the average engagement (e.g., the number of check-ins) of the users with the same coreness is positively correlated to the value of their coreness on real data [14]. Recently, the experiments in [15] find that the engagement estimation of a vertex can be more accurate if both its coreness and its position in the HCD are considered.

(Graph Visualization) The hierarchical structure depicted by HCD is an elegant visualization of a network. It helps users to better understand and explore the insights in different areas, e.g., the internet [16], biology [17], and brain networks [18].

Existing Solutions. In the construction of HCD, we should compute the coreness of every vertex, i.e., do core decomposition. It can be computed by recursively removing each vertex with the smallest degree from the graph [19]. The parallel algorithms for core decomposition are well-studied [20]–[25], and the state-of-the-art are PKC [20] and GBBS [23]. Existing works on (non-hierarchical) core decomposition such as PKC cannot build the HCD, as they cannot distinguish the connectivity among different k-cores.

(HCD Construction) To the best of our knowledge, there is no parallel solution for HCD construction. The state-of-the-art serial solution is LCPS [7] that runs in $O(m)$ time, where m is the number of edges. LCPS conducts a priority search $v_1, v_2, \cdots, v_n$, during which LCPS inserts the visited vertex one by one into a HCD under construction. Assume $c(v)$ is the coreness of v and is pre-computed for every vertex. Let R be unvisited neighbors of visited vertices $\{v_1, v_2, \cdots, v_{i-1}\}$. In the i-th iteration of LCPS, we (i) choose a vertex $v_i \in R$ with the highest priority $pri(w) = \max_{j<i, wv_j \in E}\{\min\{c(w), c(v_j)\}\}$, (ii) adjust the HCD and insert v_i into the tree nodes under processing, according to $c(v_i)$ and $pri(v_i)$, and (iii) set v_i as visited and update R along with the priority.

LCPS is still not efficient enough on large graphs and it is hard to parallelize. Δ-stepping [26] can parallelize the priority search of Dijkstra algorithm, but it cannot be applied to LCPS. If LCPS concurrently visits two vertices u and v with the same priority, the visit of u may prioritize other vertices. Thus, there will be different priority orderings in different threads, and the merged result would be incorrect due to inconsistent priorities.

(Subgraph Search on HCD) A major application of HCD is to fast search cohesive subgraphs such as the densest subgraph and the maximum clique. Given the HCD and a community scoring metric Q (e.g., average degree or clustering coefficient), the subgraph search problem on HCD is to find a k-core with the highest score regarding Q.

To the best of our knowledge, there is no parallel solution for subgraph search on HCD. The state-of-the-art serial solution is BKS [10] which incrementally computes the scores of the k-cores from $k = k_{max}$ descending to $k = 0$, with the help of a light-weight vertex ordering technique on the hierarchy. BKS is still not efficient enough on massive graphs, e.g., it may take over 5000s on a commercial machine for some community metrics. The issues with BKS in designing a parallel solution are in two folds. First, BKS computes the scores of k-cores in decreasing coreness, and it relies on the results of larger coreness. As BKS requires barriers when processing different coreness, it is not suitable for parallel execution. Second, the vertex ordering of BKS sorts all adjacency lists by coreness using a bin-sort-like method, while it is inefficient to access a bin using multiple threads.

In this paper, we aim to propose efficient parallel algorithms for HCD computation: from its fast construction to efficient search of high-quality cohesive subgraphs.

Challenges. The problems are challenging because (i) on the basis of classical core decomposition, HCD further considers the complex relations of all k-cores in the hierarchy; (ii) as proven in our paper, HCD construction is $\mathcal{P}$-complete and thus is probably inherently sequential (like linear programming); and (iii) existing methods of constructing HCD and subgraph search can hardly be parallelized, as discussed in introducing existing solutions.

We also try to partition the graph, compute the hierarchy on every partition (e.g., LCPS), then merge the results of different partitions. According to our experiments in Section V-B, parallel graph partition algorithms (e.g. Spinner [27], KaHIP [28]) are much slower than our proposed parallel algorithms for HCD construction, and an essential technique RC in the paradigm is not efficient enough. The above issues also exist for parallelizing subgraph search algorithms.

Our Solution. In this paper, we design the first parallel algorithm for HCD construction with the well-known structure of union-find [29]–[31]. As discussed above, LCPS may output incorrect results when concurrently visiting two vertices. To enable a concurrent visit of vertices, our PHCD incrementally adds the k-shell (vertices with the coreness of k) to the graph in descending k and builds the HCD in a bottom-up manner using union-find. PHCD can independently handle the vertices in the k-shell in parallel. The sequential version of PHCD is 1.24-2.33x faster than LCPS in our performance test, because the priority in LCPS is maintained in multiple dynamic arrays which are costly especially for large graphs. For the parallel version, the total number of steps in PHCD is near-linear to the number of edges in the graph.

We also propose the PBKS algorithm to efficiently search high-quality subgraphs from HCD with respect to different community metrics, e.g., average degree, conductance, modularity, and clustering coefficient. As discussed above, the

TABLE I
SUMMARY OF NOTATIONS

Symbol	Description
$G = (V, E)$	an undirected simple graph
$n; m$	number of vertices/edges (assume $m > n$)
S	a subgraph of G
$N(v)$	the neighbor set of v
$d(v)$	the degree of v in G
K_k	k-core set: the set of all (connected) k-cores of G
$c(v)$	coreness of v in G, $\max\{k \mid v \in C_k\}$
k_{max}	largest k s.t. C_k is not empty
H_k	the k-shell of G, $\{v \mid c(v) = k\}$
$r(v)$	the vertex rank of v in V
T_i	a k-core tree node in the HCD
$V(T_i)$	the set of vertices contained in T_i
$P(T_i)$	the unique parent tree node of T_i
$C(T_i)$	the children tree nodes of T_i
$tid(v)$	the id of v's belonged tree node, i.e., $v \in V(T_{tid(v)})$

existing subgraph search solution BKS [10] relies on the results of larger coreness and it needs to sort all adjacency lists by coreness. In comparison, our PBKS uses a different vertex-centric paradigm and replaces the ordering of adjacency lists with a lighter preprocessing. PBKS can process all vertices concurrently due to a new counting strategy. For every community metric based on the primary values used in the paper, we prove that PBKS is theoretically work-efficient and validate that PBKS largely outperforms BKS in practice. PBKS can also be used for the solutions to k-core related problems, e.g., densest subgraph search and finding the maximum clique.

Contributions. The major contributions are as follows.

- We first prove the $\mathcal{P}$-completeness of HCD construction and evaluate the feasibility of parallelizing existing algorithms. Then, the first parallel algorithm PHCD is proposed for HCD construction, using a novel union-find-based framework. The PHCD algorithm has a near-linear work and is practically-efficient.
- We propose the first parallel algorithm PBKS for the search of cohesive subgraph on HCD w.r.t. various community scoring metrics. We prove PBKS is work-efficient, i.e., its number of steps asymptotically matches the best sequential time complexity.
- Extensive experiments are conducted on 10 real-world graphs. For HCD construction, PHCD is up to 22x faster than the state-of-the-art serial algorithm on 40 cores, and the runtime of PHCD is close to the lower bound (overall connection cost in the union-find). For subgraph search, our PBKS is up to 50x faster than the state-of-the-art serial solution on 40 cores. We also validate that PBKS is the state-of-the-art approximate algorithm for the densest subgraph problem on both output quality and time cost.

II. Preliminaries

In this section, we introduce the definitions of core decomposition, hierarchical core decomposition, and the subgraph search problem on the HCD. Table I summarizes the notations.

A. Core Decomposition

Let $G = (V, E)$ be an undirected simple graph with $n = |V|$ vertices and $m = |E|$ edges, where we assume $m > n$.

Given a graph G, a k-core is defined as a maximal *connected* subgraph where each vertex has at least k neighbors in the subgraph [7], [8]. Let the k-core set K_k of G be the subgraph containing all (connected) k-cores. The k-core set with k from 0 to k_{max} can form a nested chain, where k_{max} (graph degeneracy) denotes the largest k such that the k-core exists.

$$K_{k_{max}} \subseteq \cdots K_1 \subseteq K_0 = V$$

Each vertex has a unique coreness $c(v) = \max\{k \mid v \in K_k\}$ which is the largest k such that v is in a k-core. The k-shell of G, denoted by $H_k = \{v \mid c(v) = k\}$, is the set of all vertices whose coreness is k. The k-shell is also the difference between the vertex set of k-core set and the $(k+1)$-core set, i.e., $V(K_k) \setminus V(K_{k+1})$. Thus, the k-core set can be induced by all c-shells with $c \geq k$, i.e., $V(K_k) = \bigcup_{c \geq k} H_c$.

The core decomposition of a graph identifies the coreness of every vertex. Batagelj-Zaversnik algorithm [19] computes core decomposition in $O(m)$ time, which recursively removes each vertex with the smallest degree in current graph.

B. Hierarchical Core Decomposition

The k-cores in a core decomposition can be organized into a hierarchy, because for every k (i) each k-core is disjoint from each other, and (ii) a k-core is contained in exactly one $(k-1)$-core.

We formalize the hierarchical structure of k-cores through HCD, where each k-core is uniquely associated with a k-core tree node in the hierarchy.

Definition 1 (k-Core Tree Node)**.** *Each k-core S in G is associated with a tree node T_S which stores the vertices of coreness k in S (i.e., $S \cap H_k$), if $S \cap H_k$ is non-empty. Such a T_S is called a k-core tree node.*

If a k-core S is associated with a k-core tree node T_S, we also say that S is the original k-core of T_S. The vertices in a k-core tree node may be separated, though its original k-core must be connected.

Definition 2 (Parent Tree Node)**.** *Given a k_1-core S_1 associated with tree node T_1 and a k_2-core S_2 associated with tree node T_2, we have T_1 is the parent of T_2, if (i) $k_1 < k_2$; (ii) $S_2 \subset S_1$; and (iii) any k'-core tree node is not the parent of T_2, where $k_1 < k' < k_2$.*

The parent-child relationships (tree edges) in the HCD record the containment and disjointness relations among all the k-cores. Similar to rebuilding the k-core set from all vertices with coreness no less than k, we can reconstruct a k-core by its associated tree node and offspring tree nodes. The HCD of a graph is defined as follows.

Definition 3 (Hierarchical Core Decomposition, i.e., HCD)**.** *Given a graph $G = (V, E)$ and the coreness $c(v)$ for every $v \in V$, the hierarchical core decomposition of G identifies (i)*

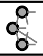
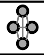
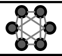
	$V(T_i)$	$P(T_i)$	$C(T_i)$
T_2		NIL	$[T_{3.1}, T_{3.2}]$
$T_{3.1}$		T_2	$[T_4]$
$T_{3.2}$		T_2	$[\,]$
T_4		$T_{3.1}$	$[\,]$

Fig. 2. The HCD Index of Figure 1(c)

all k-core tree nodes for every integer k from 0 to k_{max}; and (ii) the parent-child relationships between those tree nodes.

The state-of-the-art serial solution for constructing the HCD is LCPS with time complexity of $O(m)$ [7], which sequentially pushes a vertex according to a priority function and its unvisited neighbors into queues s.t. the subtree containing the vertex is built. Nevertheless, the algorithm is still not efficient enough for large graphs and there is no existing parallel solution.

Index Overview of HCD. A HCD is denoted by T, where T_i (id is i) is a tree node with three fields. We also use $tid(\cdot)$ to store the tree node id for every vertex.

- $V(T_i)$: the set of vertices contained in the k-core tree node, and all vertices in the tree node has coreness of k.
- $P(T_i)$: the unique parent tree node.
- $C(T_i)$: the array of children tree nodes.
- $tid(v)$: the id of the belonged tree node for every vertex v, i.e., we have $v \in V(T_{tid(v)})$.

Example 3. *Figure 1(c) depicts a HCD, and Figure 2 shows its index. The tree node T_2 is associated with the whole graph (a 2-core S_2), and it contains the 2-shell which is not connected. Each vertex only appears once in the HCD.*

C. Problem Definitions

In this paper, we aim to efficiently construct the HCD and search for cohesive subgraphs on the HCD.

- **HCD Construction.** Given a graph G, build the HCD of G that identifies all k-core tree nodes for every integer k from 0 to k_{max} and the parent-child relationships between those tree nodes.
- **Subgraph Search.** Given a graph G, the HCD of G, and a community scoring metric Q, find the k^*-core S that has the highest score among all the k-cores for any integer k with $0 \leq k \leq k_{max}$.

Given a community scoring metric, a higher score of a subgraph implies a better quality of the subgraph in some network features [32]. The subgraph search problem captures the goodness of a particular community score as well as the minimum degree constraint.

D. Community Scoring Metrics

A high-quality subgraph often has many connections inside the subgraph and few connections to the outside, which can be well measured by community scoring metrics according to different scenarios [32]. Our proposed algorithms can cover most of the metrics in [32], [33], because they can handle any (new) metric that is defined upon following primary values.

Primary Values. Let S be a subgraph to evaluate, we study the following common primary values.

- $n(S)$: the number of vertices in S;
- $m(S)$: the number of edges in S;
- $b(S)$: the number of boundary edges in S, $b(S) = |\{(u,v) \mid (u,v) \in E, u \in S, v \notin S\}|$;
- $\Delta(S)$, the number of triangles in S;
- $t(S)$, the number of triplets (three vertices connected by two edges) in S;

Metrics. Based on the above primary values, some community scoring metrics are defined as follows (normalized to be the higher the better).

- Average Degree: $f(S) = \frac{2\times m(S)}{n(S)}$ is the average degree of vertices in S;
- Internal Density: $f(S) = \frac{2\times m(S)}{n(S)\times(n(S)-1)}$ is the internal density of vertices in S;
- Cut Ratio: $f(S) = 1 - \frac{b(S)}{n(S)\times(n-n(S))}$ is the difference of 1 and the fraction of existing boundary edges over all the possible ones;
- Conductance: $f(S) = 1 - \frac{b(S)}{2\times m(S)+b(S)}$ is the difference of 1 and the proportion of degrees where the vertex points to the outside;
- Modularity: $f(P) = \sum_{i=1}^{k}\left(\frac{m(P_i)}{m} - \left(\frac{2\times m(P_i)+b(P_i)}{2\times m}\right)^2\right)$ is the modularity for a partition P on G, where P_i is the i^{th} community in the partition [34];
- Clustering Coefficient: $f(S) = \frac{3\times\Delta(S)}{t(S)}$ measures the tendency of vertices to cluster together;

Type-A/B Metrics. According to computing complexity, we divide the metrics into two categories. Type-A metric is based on $n(S)$, $m(S)$, and $b(S)$, while type-B metric is defined on high-order motifs such as $\Delta(S)$ and $t(s)$.

III. Parallel HCD Construction

In this section, we propose a scalable algorithm, named PHCD, for parallel HCD construction. Section III-A introduces the basic ideas of PHCD. Section III-B specifies the union-find in our algorithm. Section III-C computes the vertex rank in parallel, and Section III-D presents the details of PHCD. Section III-E discusses a method based on divide and conquer.

Theorem 1. *The problem of HCD construction is $\mathcal{P}$-complete.*

Proof. Given an integer k, the problem of determining if the k-core is non-empty is $\mathcal{P}$-complete [35]. This problem can be solved by determining if there is a k'-core tree node ($k' \geq k$) in the HCD. Thus, HCD construction is $\mathcal{P}$-complete. □

We prove the HCD construction problem is $\mathcal{P}$-complete, which implies the problem may be inherently sequential and cannot be effectively parallelized (like linear programming and circuit value problem). To tackle the hardness, we develop a new paradigm to compute the HCD in parallel.

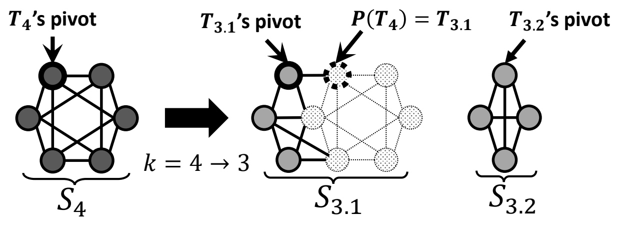


Fig. 3. The Idea of PHCD

A. Basic Idea of PHCD

As discussed in the introduction, LCPS is designed for efficient serial computation. In the i-th round, LCPS [7] chooses an unvisited neighbor v_i of visited vertices and inserts v_i into the HCD, according to a priority function. However, if LCPS concurrently visits two vertices u and v with the same priority, the visit of u may prioritize other vertices. Thus, there will be different priority orderings in different threads, and the merged result would be incorrect due to inconsistent priorities. Besides, for a divide-and-conquer paradigm discussed in Section III-E, we validate in Section V-B that it is not suitable for efficient parallelism.

From the view of efficient parallelism, we design a new framework PHCD for HCD construction. Our algorithm starts from an empty graph. Intuitively, PHCD gradually adds the k-shells to the graph in decreasing order of coreness k, during which we incrementally build the HCD in a bottom-up manner. In such a framework, each vertex in the k-shell can be processed independently in one thread. The union-find structures can be used to maintain the connectivity during the bottom-up merge. We also propose the concept of pivot to group vertices into tree nodes and identify the parent-child relationships in the HCD during the merging process. In the following, we introduce vertex rank to define pivot.

Definition 4 (Vertex Rank)**.** *A vertex v has a lower vertex rank than a vertex u, denoted by $r(v) < r(u)$, if (i) $c(v) < c(u)$, OR (ii) $c(v) = c(u)$ and $id(v) < id(u)$.*

Definition 5 (Pivot)**.** *Given a subgraph (resp. a tree node) S, the pivot of S is the vertex $u \in S$ (resp. $u \in T_S$) whose vertex rank is the lowest.*

A k-core and its associated tree node (e.g., $S_{3.1}$ and $T_{3.1}$) always have the same pivot, because the vertices in the tree node have the lowest coreness among the k-core. In a HCD, the pivot of a k-core tree node is always different from other tree nodes. We can use a pivot to uniquely identify a k-core tree node along with its original k-core.

From $k = k_{max}$ down to $k = 0$, we add the k-shell to the graph and connect them to previously added vertices (whose coreness is greater than k). In the process, we maintain the pivot of every connected component (CC) in the union-find, and we use pivot to group vertices into new tree nodes and identify the parent-child relations between tree nodes. The details are given in Section III-D.

Example 4. *Figure 3 illustrates the basic idea of using pivot on the HCD in Figure 1(c). The pivot of $T_{3.1}$ is the bold vertex (right), and it can uniquely identify $T_{3.1}$ (and $S_{3.1}$). When $k = 4 \to 3$, the pivot of CC changes from T_4's pivot (left) to $T_{3.1}$'s pivot (right) due to a lower vertex rank. There are two operations with pivot. (Group Vertices): we group the vertices in the 3-shell into new tree nodes by their pivots, e.g., $T_{3.1}$ and $T_{3.2}$ can be identified (right); (Find Parent): when $k = 4 \to 3$, the pivot of CC changes from T_4's pivot to $T_{3.1}$'s pivot, thus, $T_{3.1}$ is the parent tree node of T_4, i.e., $P(T_4) = T_{3.1}$.*

Algorithm 1: parallel vertex rank computation

```
Input  : a graph $G$, the core decomposition of $G$
Output : the vertex rank $r(v)$ for each vertex $v \in V$, the
         $k$-shell $H_k$ for every possible $k$
1  $p_{max} \leftarrow$ the number of threads;
2  distribute vertices to $V_1, \cdots, V_{p_{max}}$ in ascending vertex id;
3  for each thread $p$ from $1$ to $p_{max}$ do in parallel
4      HL[$p$] $\leftarrow$ an array with $k_{max} + 1$ bins;
5      for $v \in V_p$ in ascending vertex id do
6          atomic append $v$ to HL[$p$][$c(v)$];
7  for each $k$ from $0$ to $k_{max}$ do in parallel
8      $H_k \leftarrow$ HL[1][$k$] + HL[2][$k$] + $\cdots$ + HL[$p_{max}$][$k$];
9  $V_{sort} \leftarrow H_0 + H_1 + \cdots + H_{k_{max}}$ ;
10 for each $v \in V$ do in parallel
11     $r(v) \leftarrow$ the ranking of $v$ in $V_{sort}$;
12 return $r(v)$ for each $v \in V$;
```

B. Union-Find with Pivot

Union-find (UF) [29] can efficiently maintain the connectivity of a graph using operations such as make_set, find, union, and same_set. Besides, we define a new function get_pivot(x) that returns the pivot of x's belonged CC.

We implement the union-find with pivot as follows. For each vertex, we store three fields in the UF including parent pointer, union-find rank, and pivot $t(x)$. Let the cardinal element of a CC be r_x = find(x), where x is any vertex in the CC. The pivot of a CC is maintained at its cardinal element and we compute get_pivot(x) by returning $t(r_x)$. When calling union(x, y), r_x and r_y will be merged into a new cardinal element, and we set the element with a lower vertex rank among $t(r_x)$ and $t(r_y)$ as the pivot at the new cardinal element.

Union-find with pivot can be extended to a multi-core platform using wait-free union-find (WF-UF) [30] which is lock-free and can avoid synchronization overheads. Assume there are n elements, m UF operations, p threads, and at most F failures. WF-UF has total work of $O(n\sqrt{p} + m\alpha(n) + F)$ time, where $\alpha(n)$ is the inverse Ackermann function and $\alpha(n) < 5$ for any practical input size n.

C. Vertex Rank Computation

In Algorithm 1, we compute the vertex rank (Definition 4) in advance for better efficiency. The array V_{sort} is sorted by vertex rank and can also be used to query the k-shell H_k (set of vertices with coreness of k). Assume there are p_{max} threads available, (i) Line 2 divides all vertices by increasing id into p_{max} sets and assigns V_p to the p-th thread; (ii) for each thread, Line 3-6 further push the vertices in V_p into bins $\texttt{HL}[p][0..k_{max}]$ according to the coreness; and (iii) the concatenation of $\texttt{HL}[1...p_{max}][k]$ is H_k (Line 7-8), and the concatenation of $H_{0..k_{max}}$ is V_{sort} that sorts all vertices by vertex rank (Line 9). The total work of Algorithm 1 is $O(n)$.

D. PHCD Algorithm

Algorithm 2 presents the pseudo-code of `PHCD` algorithm that implements the paradigm of Section III-A. A line started with **atomic** is executed atomically. The HCD index is specified in the index overview of Section II. The union-find (e.g., `get_pivot` and `union`) is discussed in Section III-B.

Line 1-3 run Algorithm 1 and initialize the data structures. From $k = k_{max}$ to $k = 0$, Line 4-22 compute the HCD in four steps. Union-find (UF) with pivot in Section III-B is used to efficiently maintain the connectivity of the graph and the pivot of every CC.

- **Step 1 (Line 6-9): Find k'-Core Tree Nodes**: When constructing the tree nodes associated with k-cores, we first use pivot to identify the k'-cores tree nodes that would be merged with the k-shell vertices at Step 2 and 3, where $k' > k$ (Line 7-8). These pivots are stored in *kpc_pivot* (Line 9) and their parent k-core tree nodes will be detected in Step 4.
- **Step 2 (Line 10-12): Connectivity Check on k-Shell and k'-Cores**: In the UF, we add the k-shell to the graph and connect them to previously added vertices whose coreness is no less than k. Note that we need to visit v's each neighbor u with $c(u) \geq c(v)$ (Line 11), because the vertices may be disconnected in the tree node while connected in its associated k-core. The pivots of CCs are updated during the connection of UF (Line 12) s.t. the vertices in different tree nodes are distinguished.
- **Step 3 (Line 13-18): Create Tree Nodes Associated with k-Cores**: The vertices belonged to the same pivot are in the same tree node, so a new tree node is created for each pivot pvt with $tid(pvt) = \infty$ (Line 15-16), and $tid(v)$ is copied according to its pivot (pvt) for each $v \in H_k$ (Line 17).
- **Step 4 (Line 19-22): Find Parent Tree Nodes**: For each k'-core tree node (represented by its pivot) stored in Step 1, it is the child of a newly formed k-core tree node in Step 3. We find its parent tree node by the pivot of its belonged CC (Line 22).

Example 5. *Figure 3 shows the idea of* `PHCD`*. For $k = 4$, the 4-shell has been added and T_4 is already built. In the following, we demonstrate the process of Step 1-4 when $k = 3$.*

Step 1: visit the neighbors of the 3-shell whose coreness is greater than $k = c(v) = 3$ and uniquely store its belonged pivot into kpc_pivot. The pivot of T_4 is stored.

Step 2: add the 3-shell and link them to neighbors whose $c(u) \geq k$. There are two CCs now, and the pivot of a CC turns from T_4's pivot to $T_{3.1}$'s pivot.

Step 3: create two tree nodes and group the vertices in the 3-shell into tree nodes according to the belonged pivots.

Algorithm 2: parallel HCD construction

```
Input  : a graph $G$, the core decomposition of $G$
Output : the HCD $T$ of $G$
1  run Algorithm 1 to compute the $k$-shells and vertex rank;
2  $T \leftarrow$ an empty HCD;
3  for $v \in V$ do make_set($v$);  $tid(v) \leftarrow \infty$;
4  for $k$ from $k_{max}$ to 0 do
5      $kpc\_pivot \leftarrow \varnothing$;
6      for $v \in H_k$ do in parallel                    // Step 1
7          for $u \in N(v)$ and $c(u) > c(v)$ do
8              $pvt \leftarrow$ get_pivot($u$);
9              atomic add $pvt$ to $kpc\_pivot$ if not exists;
10     for $v \in H_k$ do in parallel                    // Step 2
11         for $u \in N(v)$ and $c(u) \geq c(v)$ do
12             union($v, u$);
13     for $v \in H_k$ do in parallel                    // Step 3
14         $pvt \leftarrow$ get_pivot($v$);
15         if $tid(pvt) = \infty$ then
16             atomic create a tree node $T_i$;  $tid(pvt) \leftarrow i$;
17         $i \leftarrow tid(v) \leftarrow tid(pvt)$;
18         atomic add $v$ to $V(T_i)$;
19     for $v \in kpc\_pivot$ do in parallel             // Step 4
20         $pvt \leftarrow$ get_pivot($v$);
21         $ch \leftarrow tid(v)$;  $pa \leftarrow tid(pvt)$;
           // $T_{pa}$ is the parent tree node of $T_{ch}$
22         $P(T_{ch}) \leftarrow T_{pa}$;  atomic add $T_{ch}$ to $C(T_{pa})$;
23 return $T$;
```

Step 4: find parent tree node for T_4 stored in Step 1. Currently, T_4's pivot belongs to a CC whose pivot is exactly $T_{3.1}$'s pivot. Therefore, $P(T_4) = T_{3.1}$.

Complexity. Assume n is the number of vertices, m is the number of edges, p threads, and $\alpha(n)$ is the inverse Ackermann function and $\alpha(n) < 5$ for any practical input size n. *(Time)* Line 1-3 run in $O(n)$ time. Line 6-12 call `get_pivot` and `union` on each edge, which needs $O(m)$ WF-UF operations. Line 13-18 visit each vertex once and need $O(n)$ WF-UF operations. Line 19-22 visit each parent-child relationship in the core forest, which calls $O(n)$ WF-UF operations. There are $O(m)$ WF-UF operations in total. Before calling Algorithm 2, we also run parallel core decomposition `PKC` [20] in $O(nk_{max} + m)$ time. Then, the overall work of Algorithm 2 is $O(n\sqrt{p} + m\alpha(n) + F)$ time, where there are at most F failures. The work of Algorithm 2 is near-linear because the dominant term $m\alpha(n) \leq 4m$ for practical n. *(Space)* The space cost of constructing HCD is $O(n)$, while the overall space complexity of Algorithm 2 is $O(m)$ due to the space of the input graph.

Correctness. Algorithm 2 can correctly compute the tree nodes in the HCD and their parent-child relationships.

Proof. We prove the correctness of Algorithm 2 by showing every step is correct.

Existing Graph. Given an integer k, the existing graph is all the k-cores if the connected components in the union-find are all k-cores. That is, for any pair of vertices v and u in the same k-core, they belong to the same set in the union-find.

Step 2. Proof by induction. Initially, the graph is empty. Assume the existing graph is all $(k+1)$-cores before the round. Then, Step 2 unites each vertex v in the k-shell with the neighbors with coreness no less than k. After that, the existing graph is all k-cores and their pivots are properly maintained.

Step 3. At this time, the existing graph is all k-cores, and the pivot of each new k-core is in the k-shell. Since the pivot can uniquely identify a new k-core, Line 15 can uniquely build a new tree node for each new k-core, and Line 17-18 can correctly group the k-shell into tree nodes by pivot.

Step 1, 4. In Step 1, Line 8 finds each k'-core ($k' > k$) that would merge into new k-cores in Step 2, and Line 9 uniquely adds its pivot to kpc_pivot. In Step 2, those pivots are merged with the k-shell. In Step 4, each $v \in kpc\_pivot$ is the pivot of a k'-core tree node $\mathcal{T}[tid\_ch]$ and it is now contained in a k-core associated with tree node $\mathcal{T}[tid\_pa]$, so they have a parent-child relationship in the HCD. □

E. Discussion on Divide and Conquer

We investigate how to compute HCD construction using divide and conquer. Steps 1-3) use existing algorithms, while Steps 4-5) can be computed by local k-core search (RC).

1) compute the coreness of every vertex.
2) divide G into p_{max} disjoint partitions $S_1, S_2, \cdots, S_{p_{max}}$.
3) run LCPS on each partition S_p to obtain the partial tree nodes in the partition.
4) obtain all k-core tree nodes of G by merging the partial results from different partitions.
5) confirm the parent-child relationships between tree nodes.

Given a vertex v, a local k-core search of v find the maximal connected subgraph containing v where all vertices have coreness no less than $c(v)$. A local k-core search of v can be computed by a BFS from v.

A local k-core search can reconstruct a k-core from its tree node. In Step 4), given a partial tree node T_*, the local k-core search starting from T_* may contain other partial tree nodes from different partitions, all these partial tree nodes (including T_*) are merged into a new tree node. In Step 5), we can confirm the parent-child relationships between two tree nodes T_i and T_j if the local k-core search from T_i contains T_j and the coreness of vertices in T_i is the lowest.

In the experiments of Section V-B, we show that the local k-core search (RC) is not efficient enough. The divide-and-conquer paradigm depends on the local k-core search, thus, it is not efficient for parallel HCD construction.

IV. Subgraph Search in Parallel

In this section, we start with the basic idea of PBKS in Section IV-A. Then, we apply PBKS to compute type-A and type-B metrics in Section IV-B and IV-C, respectively. Finally, we present the details of PBKS in Section IV-D.

The hardness of the subgraph search problem depends on the community metric and analysis task we choose. In this section, we aim to address the problem with most metrics, i.e., the metrics based on the chosen primary values.

Algorithm 3: search for best k-core on the HCD

```
Input  : a graph $G$, a community metric $Q$, the HCD $T$ of $G$
         with preprocessing in Section IV-A
Output : the $k$-core with the highest score
1  pri_val ← $[0, \dots, 0]$;
2  for each $v \in V$ do in parallel
3      $i \leftarrow tid(v)$;
4      pri_val[$i$] += $v$'s contribution to $T_i$'s primary value
       where $v$ has the lowest rank in the motif;
5  metric ← $[0, \dots, 0]$;
6  for each tree node $T_i$ from bottom to up do
7      $pa \leftarrow \{pa \mid P(T_i) = T_{pa}\}$;
8      pri_val[$pa$] += pri_val[$i$];
9      metric[$i$] ← get_metric($Q$, pri_val[$i$]);
10 $S$ ← the $k$-core with the highest score according to metric;
11 return $S$
```

A. Basic Idea of PBKS

The state-of-the-art serial solution is BKS [10] which incrementally computes the score of every k-core in decreasing k, with the help of a vertex ordering technique. The issues with BKS in designing a parallel solution are in two folds. First, BKS computes the scores of k-cores in decreasing coreness, and it relies on the results of larger coreness. As BKS requires barriers when processing different coreness, it is not suitable for parallel execution. Second, the vertex ordering of BKS sorts all adjacency lists by coreness using a bin-sort-like method, while it is inefficient to access a bin using multiple threads.

For the first issue, our PBKS computes the score in a vertex-centric manner s.t. the computation on different vertices can be independently executed. For the second issue, we replace the vertex ordering with a preprocessing that is more suitable for parallel execution, without sacrificing the effectiveness.

PBKS first computes the contributions of every vertex in parallel. The computation is vertex-centric because we uniquely count any motif (e.g., a triangle or an edge) using the vertex with the lowest vertex rank in the motif. After that, the contributions are summed up on the HCD to compute the community score. We (i) sum up the contributions of every tree node from its contained vertices, (ii) compute the primary value of every k-core by bottom-up accumulation on the HCD, and (iii) generate the community score of every k-core on the computed primary values.

Algorithm 3 presents the framework of our algorithm, where pri_val and metric store the primary values and community score of every k-core (tree node). Firstly, Line 2-4 compute primary value (motif count) of every vertex $v \in V$ where v has the lowest vertex rank, and the contributions are summed up into its belonged tree node T_i. Then, Line 6-9 sum up the primary value from bottom to up and compute the community score of every k-core. Line 6-9 can be efficiently computed by parallel tree accumulation [36]. The details are given in following subsections.

Example 6. *Let $S_{3.1} = G[S_4 + T_{3.1}]$ be the 3-core in Figure 1(c). Assume we need to compute the number of vertices in $S_{3.1}$, i.e., $n(S_{3.1})$. We have $n(S_4) = 6$. For any $v \in V(T_{3.1})$, we compute the contribution $\Delta n(v) = 1$. The contributions of $T_{3.1}$ is $\Delta n(T_{3.1}) = \sum_{v \in V(T_{3.1})} \Delta n(v) = 3$. Then, we can compute the primary value of $S_{3.1}$ based on the children of $T_{3.1}$ in the HCD, i.e., $n(S_{3.1}) = n(S_4) + \Delta n(T_{3.1}) = 6 + 3 = 9$. Similarly, we can incrementally count the number of vertices and edges of every k-core in a bottom-up manner, and then compute the average degree of every k-core.*

Preprocessing on the HCD. During the score computation, we always ask the number of neighbors with less (greater, or equal) coreness, e.g., $|\{u \mid u \in N(v) \wedge c(u) > c(v)\}|$. To answer such queries efficiently, the preprocessing is executed once before any score computation. For each vertex v in parallel, we pre-compute the number of neighbors with greater coreness than v and with the same coreness to v. Then, we can answer the number of neighbors with less (greater, or equal) coreness instantly. The preprocessing can be done in $O(m)$ work. When computing subgraph search using different metrics, the preprocessing can speed up the computation.

B. Parallel Type-A Metric Computation

Algorithm 4 applies the framework in Algorithm 3 to type-A score computation. We store the the primary values (the number of vertices, edges, and boundary edges) of every vertex in arrays v,e, and b. The score of every k-core is stored in metric.

Line 2-9 count the contribution of every vertex to the primary value of tree nodes, where the vertex has the lowest vertex rank. Here, *gt_k*, *eq_k*, and *lt_t* represent the number of v's neighbors whose coreness is greater, equal, or less than $c(v)$, respectively. Each vertex contributes (i) 1 new vertex; (ii) $gt\_k + \frac{1}{2} eq\_k$ new edges, where *eq_k* is halved because it would be counted by both endpoints; and (iii) $lt\_k - gt\_k$ boundary edges, because v has *lt_k* neighbors outside $c(v)$-core and *gt_k* neighbors inside. Then, Line 11-14 do a bottom-up tree accumulation on v,e,b and compute the community score of every k-core.

Complexity. After $O(m)$ work of preprocessing (executed once for different metrics), the total work of Algorithm 4 is $O(n)$, since we visit each vertex and tree node once with a constant number of counting. Algorithm 4 is work-efficient because we need at least $O(m)$ time for a metric requiring the number of edges.

C. Parallel Type-B Metric Computation

Algorithm 5 follows the paradigm of Algorithm 3, and it can efficiently compute the scores for type-B metrics. Array ta and tp store the number of triangles and triplets in every k-core (tree node), while array metric stores the community scores of k-cores. Line 2-15 count the number of triangles and triplets contributed by every vertex. Line 17-20 do bottom-up tree accumulation on the numbers and compute the community score. Line 22 returns the best k-core.

Algorithm 4: search for best k-core w.r.t. a type-A metric

```
Input  : a graph $G$, a community metric $Q$, the HCD $T$ of $G$
         with preprocessing in Section IV-A
Output : the $k$-core with the highest score
1  v $\leftarrow$ e $\leftarrow$ b $\leftarrow [0, \dots, 0]$;
2  for each $v \in V$ do in parallel
       /* answer Line 3-5 by preprocessing    */
3      $gt\_k \leftarrow |\{u \mid u \in N(v) \wedge c(u) > c(v)\}|$ ;
4      $eq\_k \leftarrow |\{u \mid u \in N(v) \wedge c(u) = c(v)\}|$;
5      $lt\_k \leftarrow |N(v)| - gt\_k - eq\_k$;
6      $i \leftarrow tid(v)$;
7      v[$i$] += 1;
8      e[$i$] += $gt\_k + \frac{1}{2} \cdot eq\_k$;
9      b[$i$] += $lt\_k - gt\_k$;
10 metric $\leftarrow [0, \dots, 0]$;
11 for each tree node $T_i$ from bottom to up do
12     $pa \leftarrow tid(P(T_i))$;
13     v[$pa$] += v[$i$]; e[$pa$] += e[$i$]; b[$pa$] += b[$i$];
14     metric[$i$] $\leftarrow$ get_metric($Q$, v[$i$], e[$i$], b[$i$]);
15 $S \leftarrow$ the $k$-core with the highest score according to metric;
16 return $S$
```

Triangle Counting. Line 2-7 count the number of triangles contributed by v. We enumerate each edge (v, u) where u has less (or equal) degree than v. Then, for each u's neighbor w, we check if edge (v, w) exists. If so, (v, u, w) is a triangle that is newly added when $k = c(w)$, where w has the lowest vertex rank among three endpoints, and we uniquely count it in ta array.

Triplet Counting. Line 8-15 count the number of triplets contributed by v. A triplet has three vertices connected by two edges. Line 8-15 consider all triplets centered at v. Line 9 counts the triplets newly added when $k = c(v)$, by choosing two neighbors of v with coreness no less than v. For triplets newly added when $c(w) = k < c(v)$, Line 10-15 either (i) choose two neighbors with coreness k by $\binom{cnt\_k}{2}$, or (ii) choose a neighor with coreness k and another with coreness greater than k by $gt\_k \times cnt\_k$.

Complexity. *(Triangle)* Line 3-7 iterate over each edge (u, v) where Line 4 guarantees u has lower degree than v. Thus, the number of operations in Line 3-7 is bounded by the number of visited w which is $\sum_{(u,v) \in E} \min\{d(u), d(v)\} = O(m^{1.5})$. *(Triplet)* Line 8-9 need $O(m)$ time for visiting all neighbor sets. Line 10-13 cost $d(v)$ operations, because we can iterate over the neighbor set to pre-compute the neighbors with coreness of k $N(v) \cap H_k$ and a vertex w in it. For each vertex v, Line 14-15 require $c(v) \le d(v)$ operations. The time complexity of Line 14-15 is $\sum_{v \in V} d(v) = 2m = O(m)$. *(Overall)* Line 17-20 cost $O(|T|)$ time, where $|T| \le n$. Overall, the total work of Algorithm 5 is $O(m^{1.5})$. For any metric requiring triangle counting, the cost is at least $O(m^{1.5})$ time. Consequently, Algorithm 5 is work-efficient.

Correctness. A motif is added exactly when the endpoint with the lowest vertex rank (and coreness) is added. *(Triangle)* Line 4 visits each edge (u, v) exactly once. Conditions $w \in N(u)$ and $w \in N(v)$ find the common neighbors of (u, v) that forms

Algorithm 5: search for best k-core w.r.t a type-B metric

```
Input  : a graph $G$, a community metric $Q$, the HCD $T$ of $G$
         with preprocessing in Section IV-A
Output : the $k$-core with the highest score
1  ta ← tp ← $[0, \dots, 0]$;
2  for each $v \in V$ do in parallel
3      for each $u \in N(v)$ do in parallel
4          if $d(u) < d(v)$ or $(d(u) = d(v)$ and $u < v)$ then
5              for each $w \in N(u)$ do
6                  if $w \in N(v)$ and $w$ has the lowest vertex
                   rank among $u, v, w$ then
7                      atomic ta$[tid(w)]$ += 1;
       /* answer Line 8 by preprocessing */
8      $gt\_k \leftarrow |\{u \mid u \in N(v) \wedge c(u) \geq c(v)\}|$;
9      tp$[tid(v)]$ += $\binom{gt\_k}{2}$;
10     for each $k$ from $c(v) - 1$ down to 0 do
11         $cnt\_k \leftarrow |N(v) \cap H_k|$;
12         if $cnt\_k > 0$ then
13             $w \leftarrow$ any vertex in $N(v) \cap H_k$;
14             atomic tp$[tid(w)]$ += $\binom{cnt\_k}{2} + gt\_k \times cnt\_k$;
15             $gt\_k$ += $cnt\_k$;
16 metric ← $[0, \dots, 0]$;
17 for each tree node $T_i$ from bottom to up do
18     $pa \leftarrow tid(P(T_i))$;
19     ta$[pa]$ += ta$[i]$,  tp$[pa]$ += tp$[i]$;
20     metric$[i]$ ← get_metric$(Q,$ ta$[i]$, tp$[i])$;
21 $S \leftarrow$ the $k$-core with the highest score according to metric;
22 return $S$
```

a triangle. Each triangle is checked three times by Line 6, but the vertex with the lowest vertex rank only occurs as w once, so we can uniquely count it using Line 7. *(Triplet)* The triplets containing v are added when $k \leq c(v)$. In case $k = c(v)$, the correctness of Line 10-15 is immediate. In case $k < c(v)$, v has 1-2 neighbors whose coreness equals k, and Line 10-15 covers the two cases correctly.

D. PBKS: Parallel Subgraph Search

Before the subgraph search, we need to execute parallel core decomposition using the state-of-the-art PKC [20] and our proposed parallel algorithm PHCD (Algorithm 2) for HCD construction. We then execute PBKS as follows.

1) *Preprocessing (Section IV-A):* the preprocessing is executed once to quickly answer the number of neighbors with less (or greater, or equal) coreness.
2) *Score Computation (Section IV-B and IV-C):* we use Algorithm 4 to compute type-A metric, and Algorithm 5 to compute type-B community metric.

The algorithm returns a k-core with the highest community score. PBKS may benefit the solution to k-core related problems, e.g., finding the densest subgraph, the maximum clique, and the size-constrained k-core. For instance, the k-core with the highest average degree is a 0.5-approximate solution to densest subgraph problem, because k_{max}-core is one of our candidate results which is a 0.5-approximate solution [37].

TABLE II
STATISTICS OF DATASETS

Dataset	n	m	d_{avg}	k_{max}	$\lvert T \rvert$
As-Skitter	1,696,415	11,095,298	13.1	111	902
LiveJournal	3,997,962	34,681,189	17.3	360	1755
Hollywood	1,069,126	56,306,653	105.3	2208	678
Orkut	3,072,441	117,185,083	76.3	253	253
Human-Jung	784,262	267,844,669	683.0	1200	4087
Arabic-2005	22,744,080	639,999,458	56.3	3247	28693
IT-2004	41,291,594	1,150,725,436	55.7	3224	53023
Friend**S**ter	65,608,366	1,806,067,135	55.1	304	450
SK-2005	50,636,154	1,949,412,601	77.0	4510	14356
UK-2007-05	105,896,555	3,738,733,648	70.6	5704	79318

Complexity. After $O(m)$ work of preprocessing, PBKS can compute type-A metric in $O(n)$ work and type-B metric in $O(m^{1.5})$ work. The score computation are both work-efficient.

V. EXPERIMENTAL EVALUATION

In this section, we conduct extensive experiments on real-world networks to evaluate the performance of our proposed parallel algorithms.

A. Experimental Setup

Datasets. The statistics of datasets are shown in Table II in ascending order of the number of edges. All directed datasets are symmetrized in the experiments. Hollywood and Human-Jung are from NetworkRepository[1], Arabic-2005, IT-2004, SK-2005, and UK-2007-05 are from LAW Datasets[2], and the others are from SNAP Datasets[3]. The dataset abbreviations are highlighted in bold. k_{max} is the largest vertex coreness in the graph, and $|T|$ is the number of tree nodes in the HCD.

Algorithms. The major algorithms evaluated in our experiments are as follows. The algorithms for comparison are all state-of-the-art.

1) Core Decomposition
   - PKC: the parallel core decomposition PKC [20]. As Graph Based Benchmark Suite (GBBS) [23] can be an alternative, we report the smaller runtime from PKC and GBBS for each test. We use the open-source code.
2) HCD Construction
   - LCPS: the serial algorithm for HCD construction [7].
   - PHCD: our method for parallel HCD construction proposed in Section III-D.
3) Subgraph Search
   - BKS: the serial algorithm for finding the best k-core on HCD [10].
   - PBKS: our framework for parallel subgraph search proposed in Section IV-D.

Environment. Our experiments are performed on a 40-core server with 2× Intel Xeon 20-core CPU (E5-2630 v4 @ 2.2GHz) and 128G memory. All algorithms are implemented in C++ using GCC 10.2.0 and -O3 optimization. We use OpenMP for shared memory multiprocessing programming.

[1]http://networkrepository.com
[2]http://law.di.unimi.it/datasets.php
[3]http://snap.stanford.edu/data

TABLE III
TIME COST OF HCD CONSTRUCTION.

Dataset	(1)			(40)		
	PHCD (s)	LB	LCPS	PHCD (s)	LB	RC
AS	0.300	0.30x	1.66x	0.071	0.55x	4.06x
LJ	1.272	0.36x	1.24x	0.197	0.65x	9.11x
H	0.700	0.47x	1.71x	0.125	0.57x	63.82x
O	2.518	0.35x	1.37x	0.447	0.28x	25.35x
HJ	2.224	0.48x	2.05x	0.296	0.37x	124.97x
A	5.808	0.48x	1.87x	1.208	0.44x	9.46x
IT	10.885	0.44x	1.84x	1.766	0.54x	13.37x
FS	90.730	0.54x	2.12x	8.778	0.77x	58.74x
SK	16.372	0.50x	2.33x	2.609	0.63x	20.23x
UK	37.580	0.43x	2.02x	5.299	0.52x	22.43x

LB is the lower-bound cost using union-find, RC is the cost of local k-core search, where (1) is the serial runtime, and (40) is the 40-core time. We report the runtime of PHCD in seconds, and PHCD's relative speedup to other algorithms.

B. Performance of HCD Construction

In the following, we report the runtime performance of PHCD and compare it with other related algorithms.

Serial Performance. In the (1) column of Table III, we compare the serial performance of PHCD with the state-of-the-art (LCPS) by reporting PHCD's relative speedup to LCPS. The serial PHCD is more efficient than LCPS on all graphs (speedup ratio is 1.24-2.33x) and the performance gap is enlarged on larger graphs. This is because that the priority function in LCPS is maintained in multiple dynamic arrays whose constant time cost is high and would increase with the growth of network size. In contrast, the union-find cost in PHCD scales stably. The runtime is also affected by network structure, e.g., FriendSter has about half the size of UK-2007-05 but requires much more time. The larger cost may come from the frequent pivot update in FriendSter, which is caused by small tree node numbers and giant components.

Comparison to Lower-bound Cost. LB unions every adjacent vertex pair, and it is a lower-bound cost for a union-find-based algorithm. Table III reports PHCD's relative speedup to LB on 1 core and 40 cores. For the serial case, PHCD's speedup ratio to LB is about 0.4 on large graphs; For the 40-core case, PHCD's speedup ratio to LB is about 0.5 on most datasets. As PHCD has a close performance to the lower bound, it achieves satisfactory results under the union-find-based paradigm.

Feasibility on Divide and Conquer. In the following, we show that it is infeasible to design an efficient parallel algorithm using the divide-and-conquer method in Section III-E.

Firstly, recent algorithms for parallel graph partitioning (e.g. Spinner [27], KaHIP [28]) are much slower than our PHCD. Specifically, on a graph with about 2 billion edges (SK-2005), KaHIP requires about 200 seconds on 2000 cores, while PHCD only costs about 2.6 seconds using 40 cores. This issue also applies to Spinner that requires about 100 seconds on 40 cores.

Secondly, a divide-and-conquer paradigm depends on the local k-core search (RC in Section III-E) which is not efficient enough. We test RC by computing the parent-child relations in the HCD. RC (40) column of Table III reports the speedup of PHCD to RC on 40 cores. We find that PHCD is 4.06x-124.97x faster than RC, and PHCD outperforms RC by one order of magnitude on UK which has over 3 billion edges.

Improvement over LCPS. In Figure 4, we report the relative speedup ratio of PHCD compared with LCPS. When 40 cores are used, PHCD is up to 22x faster than LCPS. On all the datasets, PHCD scales well as the number of threads grows. We also find PHCD has a better speedup ratio on larger networks. The results show that our proposed PHCD is practically scalable and efficient on real-world networks.

In Figure 5, we further include the cost of computing the input, i.e., core decomposition for PHCD and LCPS. The overall cost including the runtime of core decomposition is reported. Compared with Figure 4, the speedup ratio is slightly reduced because PKC has a lower speedup than PHCD.

C. Performance of Subgraph Search

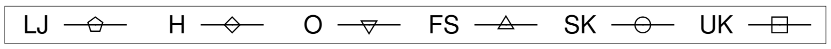
Compared with the BKS, our PBKS significantly accelerates the subgraph search without sacrificing its effectiveness.

Approximate Densest Subgraph. The densest subgraph is the subgraph with the largest average degree [37]. Let PBKS-D be our algorithm that returns the k-core with the highest average degree. PBKS-D can return an approximate solution for densest subgraph search with a 0.5 approximation ratio, because its result is always better than the k_{max}-core which is a 0.5-approximate solution [37]. In Table IV, we compare PBKS-D with a recent approximate solution CoreApp [37] and the state-of-the-art Opt-D [10] based on BKS. PBKS-D outputs the same result to Opt-D with a largely improved efficiency. Therefore, PBKS-D is the state-of-the-art approximate solution for densest subgraph problem on both output quality (average degree) and time cost.

Maximum Clique. The maximum clique is the largest subset of vertices where every vertex pair is adjacent [38]. Let S^* be the output subgraph of PBKS-D. Table IV shows that S^* contains the maximum clique with high probability (7 out of 10 datasets), even though the size of S^* is only 0.005%-1.147% of the whole vertex set. Our method can be a promising pruning technique for finding the maximum clique.

Speedup of Subgraph Search. Table V compares the subgraph search of the 40-core PBKS and the serial BKS (in PBKS's speedup). Figure 6 and 8 report the speedup of PBKS's score computation using different cores. For type-A metrics, the speedup ratio can be up to 50x because our PBKS has a better algorithm design. For type-B metrics, PBKS can achieve about 20x speedup when 40 cores are used.

Runtime with Computing the Input. When taking a graph as the input, the time cost of subgraph search further includes the runtime of core decomposition and HCD construction with preprocessing. Figure 7 and 9 report the speedup ratio of total runtime, i.e., PKC + PHCD + PBKS's speedup to PKC + LCPS + BKS. We achieve a better speedup on harder cases, i.e., on type-B metrics, because the score computation dominates the runtime in this case. Compared with Figure 6 and 8 (score computation), the speedup ratio here is reduced, because the speedup ratio of computing the inputs is smaller than PBKS.



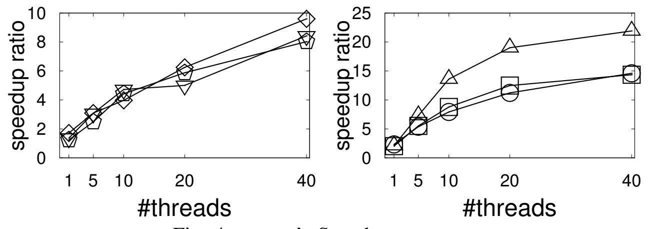


Fig. 4. `PHCD`'s Speedup to `LCPS`

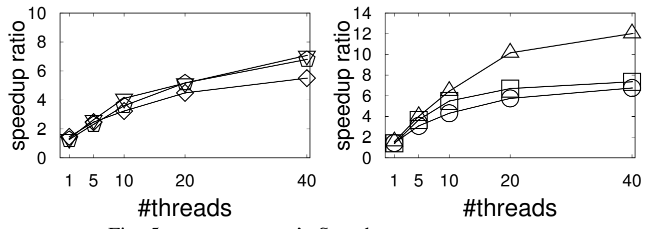


Fig. 5. `PKC` + `PHCD`'s Speedup to `PKC` + `LCPS`

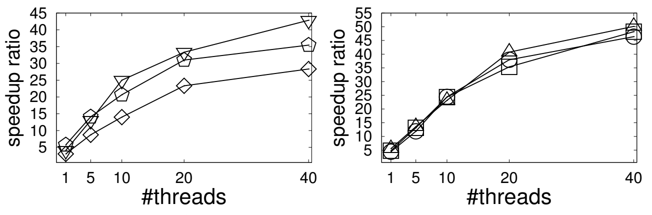


Fig. 6. `PBKS`'s Speedup to `BKS` (Type-A)

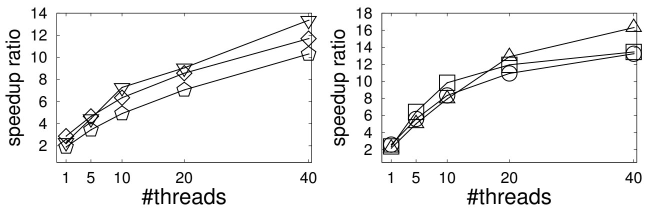


Fig. 7. `PKC` + `PHCD` + `PBKS`'s Speedup to `PKC` + `LCPS` + `BKS` (Type-A)

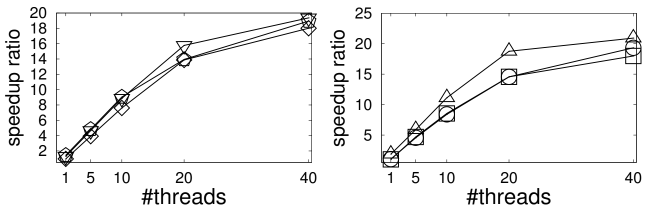


Fig. 8. `PBKS`'s Speedup to `BKS` (Type-B)

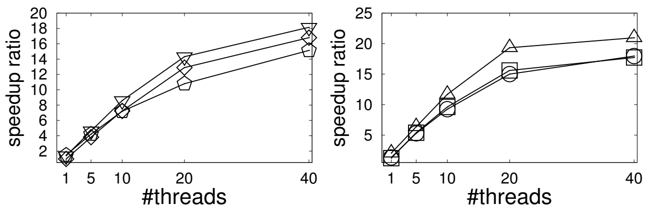


Fig. 9. `PKC` + `PHCD` + `PBKS`'s Speedup to `PKC` + `LCPS` + `BKS` (Type-B)

TABLE IV
THE PERFORMANCE OF `PBKS-D` ON DENSEST SUBGRAPH & MAXIMUM CLIQUE

Dataset	`CoreApp`		`Opt-D`	`PBKS-D`		`PBKS-D` (output S^*)	
	d_{avg}	time (s)	time (s)	d_{avg}	time (s)	$MC \subseteq S^*$	$\lvert S^*\rvert/n$
AS	150.02	1.145	1.374	**178.801**	**0.196**		0.027%
LJ	374.71	4.943	4.832	**387.027**	**0.529**	✓	0.011%
H	**2208**	3.002	3.635	**2208**	**0.542**	✓	0.207%
O	438.64	20.14	11.72	**455.732**	**1.159**		0.854%
HJ	2013.88	15.272	14.457	**2114.915**	**2.851**	✓	1.147%
A	3247	40.703	35.359	**3248.92**	**4.511**	✓	0.014%
IT	3238.921	90.86	77.276	**4016.37**	**8.036**	✓	0.010%
FS	513.85	1041.528	836.279	**547.035**	**30.022**		0.08%
SK	4513.00	202.682	125.04	**4514.99**	**12.890**	✓	0.009%
UK	5704	300.67	299.186	**5704.99**	**24.243**	✓	0.005%

d_{avg} is the average degree of the output subgraph. The d_{avg} of `Opt-D`'s output is equal to `PBKS-D`. S^* is the output subgraph of `PBKS-D`. $MC \subseteq S^*$ means the maximum clique is contained in S^*. $|S^*|/n$ is the vertex proportion of S^* in the whole graph.

TABLE V
RUNTIME OF SUBGRAPH SEARCH

Dataset	Type-A (s)		Type-B (s)	
	(40)	(1)	(40)	(1)
AS	0.004	25.00x	0.176	24.49x
LJ	0.009	34.44x	0.892	18.94x
H	0.003	23.33x	5.400	18.01x
O	0.007	42.86x	5.320	19.36x
HJ	0.004	20.00x	45.406	19.93x
A	0.053	31.47x	14.144	15.46x
IT	0.083	38.05x	22.706	18.24x
FS	0.139	50.13x	230.711	19.92x
SK	0.090	46.39x	37.216	17.33x
UK	0.193	48.20x	112.892	15.99x

(40) reports the 40-core runtime of `PBKS` in seconds, and (1) reports `PBKS`'s relative speedup to the serial `BKS`.

Speedup of Each Component. Figure 10 shows the speedup ratio of each component with `PBKS` when 40 cores are used, compared to the serial performance with `BKS`. CD is colored in grey because we use existing solutions [20], [23], while other components are proposed in our paper. For CD, parallel core decomposition has the lowest speedup ratio among all components due to the hardness of parallelism; For HCD, the time cost of the construction algorithm would limit the overall efficiency if it is not parallelized by our `PHCD`; For SC-A, type-A score computation of `PBKS` is up to 50x faster than `BKS`. The speedup ratio is over 40 as the proposed computing framework of `PBKS` is effective. For SC-B, although the computation of higher-order motifs is hard for parallelism, type-B score computation of `PBKS` is up to 20x faster than `BKS`'s counterpart.

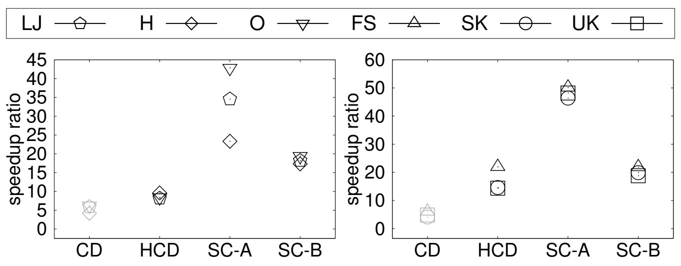


Fig. 10. `PBKS`'s Speedup to `BKS` by Component (40-core).

CD: Core Decomposition; HCD: Hierarchical Core Decomposition; SC-A: Type-A Score Computation (Excluding Preprocessing Time).

VI. EXTENSIONS OF OUR SOLUTIONS

(Novel Subgraph Search Task) As reported in Section V-B, `PBKS` is effective in the solution to approximate densest

subgraph search and maximum clique. PBKS can also facilitate other novel tasks that search for high-quality communities with both degree constraint and particular community score guarantee. Users can apply the framework with other promising subgraph search strategies, e.g., using new or assembled community scoring metrics.

(Efficient Subgraph Index) The HCD can compress the hierarchy of k-cores in $O(n)$ space. Such a structure can facilitate the solutions to many k-core related subgraph search problems. For example, the structure has been applied to construct the index for querying influential communities [11] and attributed communities [12]. After the index construction, the desired communities may be answered time-optimally.

(Finding the Best k) We can find the best parameter k for k-core decomposition by computing the k-core set with the highest scores on certain metrics [10]. Based on the paradigm in Section IV, we (i) concurrently compute the contribution of each vertex and obtain the primary value of every k-core set, (ii) compute the community score of every k-core set from primary value, and (iii) choose the k with the highest score.

(Other Cohesive Subgraph Model) Inspired by the framework of PHCD and PBKS, we can propose parallel hierarchy construction algorithms and parallel subgraph search algorithms for other cohesive subgraph models with a hierarchical decomposition, such as k-truss [39] and k-ECC [40].

VII. Related Works

Cohesive Subgraph Search. Different cohesive subgraph models are studied in the literature including clique [41], quasi-clique [42], nucleus [43], [44], k-core [8], [19], [45], k-truss [39], [46], [47], k-plex [48], and k-ecc [40], [49]. Some of these models can also decompose a graph into a hierarchical structure, e.g., core decomposition [19], [50], [51], truss decomposition [47], [52], [53], and ecc decomposition [40], [54]. Cohesive subgraph search is surveyed in [55]–[57], and many methods can be applied to other graph models, such as bipartite graph [58], [59] and heterogeneous information networks [60], [61]. The challenges of processing large graphs are summarized in [62]. Some community detection methods can also generate the hierarchical relationships among communities, e.g., label propagation [63] and Louvain [64].

k-Core and Core Decomposition. The CD algorithm [19] can compute core decomposition in $O(m)$ time. The efficient implementation of core decomposition on a single PC is studied in [45]. EM-Core [51] is an external core decomposition algorithm that adopts a top-down paradigm. In case the space limit is linear to the number of vertices, I/O efficient core decomposition [50] can be used. The streaming core decomposition is proposed in [65]. The variants of k-core model are often used to find high-quality communities under different scenes, e.g., (k,r)-core [66], influential k-core [11], radius-bounded k-core [67], persistent k-core [68], and skyline k-core [69]. Core decomposition has been widely applied to various areas such as community discovery [66], [68]–[70], influential spreader identification [71], [72], network analysis [16], [18], [73], anomaly detection [73], evaluating contagion power of vertices [71], [74], and graph visualization [1], [75].

Distributed/Parallel Core Decomposition. MPM algorithm [21] is a distributed core decomposition algorithm that converges in $it_{\text{MCM}} < k_{max} \ll n$ rounds and runs in $O(it_{\text{MCM}}m)$ time. ParK [24] is a shared-memory parallel algorithm for k-core decomposition. Based on ParK algorithm, PKC [20] adds more optimization techniques and has a lower synchronization overhead, and thus achieves a better performance. Both ParK and PKC algorithms run in $O(nk_{max}+m)$ time. Using a single machine with a terabyte of RAM and Julienne framework [22], parallel k-core decomposition can be scaled to a graph with 128 billion edges [23]. A $(2+\delta)$-approximate algorithm for k-core decomposition is proposed in [25]. The above studies focus on the computation of vertex coreness and cannot be used to compute the HCD.

Hierarchical Core Decomposition. LCPS [7] can build the HCD of a graph in $O(m)$ time. The algorithm of HCD maintenance on large dynamic networks is proposed in [15]. HCD can be used to find the best k-core, and it can help solve k-core related problems, e.g., finding the densest subgraph, the maximum clique, and the size constrained k-core [10].

ShellStruct [76] is a structure equivalent to HCD, which is proposed to efficiently handle the local query of k-cores. CL-Tree [12] is used to efficiently search attributed communities, and it contains a data structure equivalent to HCD. Compared with above serial algorithms, our PHCD puts emphasis on better parallelism, i.e., computational steps with higher cohesion, less data race, and simpler data structure.

ICP-Index [11] is a HCD-like structure that can answer influential communities time-optimally. The hierarchy of nucleus decomposition generalizes the k-core and k-truss decomposition [43], and it can compute the HCD. A parallel solution for local nucleus query (returning the nucleus containing a given vertex) is proposed in [44], but there is no parallel solution for the hierarchy construction of nucleus decomposition. The above community search index algorithms [12], [43], [76] related to HCD construction are all serial, and they can be more efficient if parallelized.

VIII. Conclusion and Future Work

The HCD has various applications on real-world networks, but the existing solutions cannot efficiently handle massive graphs. Despite the $\mathcal{P}$-completeness of HCD construction, we develop parallel algorithms that are efficient in both theory and practice. We also propose the first parallel algorithm for searching high-quality cohesive subgraphs from the HCD, regarding different community scoring metrics, where the score computation is work-efficient for all the studied metrics. Extensive experiments are conducted on 10 networks with up to billions of edges. The results confirm that our proposed parallel algorithms largely outperform the existing solutions. To the best of our knowledge, the approximation guarantees of k-core on optimizing the metrics other than average degree are still unclear. It is interesting to investigate the possible approximation guarantees for these metrics in the future.

References

[1] F. Zhao and A. K. H. Tung, "Large scale cohesive subgraphs discovery for social network visual analysis," *PVLDB*, vol. 6, no. 2, pp. 85–96, 2012.

[2] X. Chen, K. Wang, X. Lin, W. Zhang, L. Qin, and Y. Zhang, "Efficiently answering reachability and path queries on temporal bipartite graphs," *PVLDB*, vol. 14, no. 10, pp. 1845–1858, 2021.

[3] J. Kim, T. Guo, K. Feng, G. Cong, A. Khan, and F. M. Choudhury, "Densely connected user community and location cluster search in location-based social networks," in *SIGMOD*, 2020, pp. 2199–2209.

[4] Z. Yang, L. Lai, X. Lin, K. Hao, and W. Zhang, "Huge: An efficient and scalable subgraph enumeration system," in *SIGMOD*, 2021, pp. 2049–2062.

[5] S. Wang and Y. Tao, "Efficient algorithms for finding approximate heavy hitters in personalized pageranks," in *SIGMOD*, 2018, pp. 1113–1127.

[6] M. Altaf-Ul-Amine, K. Nishikata, T. Korna, T. Miyasato, Y. Shinbo, M. Arifuzzaman, C. Wada, M. Maeda, T. Oshima, H. Mori *et al.*, "Prediction of protein functions based on k-cores of protein-protein interaction networks and amino acid sequences," *Genome Informatics*, vol. 14, pp. 498–499, 2003.

[7] D. W. Matula and L. L. Beck, "Smallest-last ordering and clustering and graph coloring algorithms," *J. ACM*, vol. 30, no. 3, pp. 417–427, 1983.

[8] S. B. Seidman, "Network structure and minimum degree," *Social networks*, vol. 5, no. 3, pp. 269–287, 1983.

[9] A. Clauset, C. Moore, and M. E. Newman, "Hierarchical structure and the prediction of missing links in networks," *Nature*, vol. 453, no. 7191, p. 98, 2008.

[10] D. Chu, F. Zhang, X. Lin, W. Zhang, Y. Zhang, Y. Xia, and C. Zhang, "Finding the best k in core decomposition: A time and space optimal solution," in *ICDE*. IEEE, 2020, pp. 685–696.

[11] R.-H. Li, L. Qin, J. X. Yu, and R. Mao, "Influential community search in large networks," *PVLDB*, vol. 8, no. 5, pp. 509–520, 2015.

[12] Y. Fang, R. Cheng, S. Luo, and J. Hu, "Effective community search for large attributed graphs," *PVLDB*, vol. 9, no. 12, pp. 1233–1244, 2016.

[13] F. D. Malliaros and M. Vazirgiannis, "To stay or not to stay: modeling engagement dynamics in social graphs," in *CIKM*, 2013, pp. 469–478.

[14] Q. Linghu, F. Zhang, X. Lin, W. Zhang, and Y. Zhang, "Global reinforcement of social networks: The anchored coreness problem," in *SIGMOD*, 2020, pp. 2211–2226.

[15] Z. Lin, F. Zhang, X. Lin, W. Zhang, and Z. Tian, "Hierarchical core maintenance on large dynamic graphs," in *PVLDB*, vol. 14(5), 2021, pp. 757–770.

[16] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani, "K-core decomposition of internet graphs: hierarchies, self-similarity and measurement biases," *NHM*, vol. 3, no. 2, pp. 371–393, 2008.

[17] F. Morone, G. Del Ferraro, and H. A. Makse, "The k-core as a predictor of structural collapse in mutualistic ecosystems," *Nature Physics*, vol. 15, no. 1, p. 95, 2019.

[18] M. Daianu, N. Jahanshad, T. M. Nir, A. W. Toga, C. R. J. Jr., M. W. Weiner, and P. M. Thompson, "Breakdown of brain connectivity between normal aging and alzheimer's disease: A structural *k*-core network analysis," *Brain Connectivity*, vol. 3, no. 4, pp. 407–422, 2013.

[19] V. Batagelj and M. Zaversnik, "An o(m) algorithm for cores decomposition of networks," *CoRR*, vol. cs.DS/0310049, 2003.

[20] H. Kabir and K. Madduri, "Parallel k-core decomposition on multicore platforms," in *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPS Workshops)*. IEEE Computer Society, 2017, pp. 1482–1491.

[21] A. Montresor, F. D. Pellegrini, and D. Miorandi, "Distributed k-core decomposition," *TPDS*, vol. 24, no. 2, pp. 288–300, 2013.

[22] L. Dhulipala, G. Blelloch, and J. Shun, "Julienne: A framework for parallel graph algorithms using work-efficient bucketing," in *SPAA*, 2017, pp. 293–304.

[23] L. Dhulipala, G. E. Blelloch, and J. Shun, "Theoretically efficient parallel graph algorithms can be fast and scalable," in *SPAA*, C. Scheideler and J. T. Fineman, Eds. ACM, 2018, pp. 393–404.

[24] N. S. Dasari, D. Ranjan, and M. Zubair, "Park: An efficient algorithm for k-core decomposition on multicore processors," in *2014 IEEE International Conference on Big Data*. IEEE, 2014, pp. 9–16.

[25] Q. C. Liu, J. Shi, S. Yu, L. Dhulipala, and J. Shun, "Parallel batch-dynamic *k*-core decomposition," *arXiv preprint arXiv:2106.03824*, 2021.

[26] U. Meyer and P. Sanders, "δ-stepping: a parallelizable shortest path algorithm," *Journal of Algorithms*, vol. 49, no. 1, pp. 114–152, 2003.

[27] C. Martella, D. Logothetis, A. Loukas, and G. Siganos, "Spinner: Scalable graph partitioning in the cloud," in *ICDE*. Ieee, 2017, pp. 1083–1094.

[28] H. Meyerhenke, P. Sanders, and C. Schulz, "Parallel graph partitioning for complex networks," *TPDS*, vol. 28, no. 9, pp. 2625–2638, 2017.

[29] R. E. Tarjan and J. Van Leeuwen, "Worst-case analysis of set union algorithms," *J. ACM*, vol. 31, no. 2, pp. 245–281, 1984.

[30] R. J. Anderson and H. Woll, "Wait-free parallel algorithms for the union-find problem," in *STOC*, 1991, pp. 370–380.

[31] S. V. Jayanti and R. E. Tarjan, "Concurrent disjoint set union," *Distributed Computing*, pp. 1–24, 2021.

[32] T. Chakraborty, A. Dalmia, A. Mukherjee, and N. Ganguly, "Metrics for community analysis: A survey," *CSUR*, vol. 50, no. 4, pp. 54:1–54:37, 2017.

[33] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *KAIS*, vol. 42, no. 1, pp. 181–213, 2015.

[34] M. E. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical review E*, vol. 69, no. 2, p. 026113, 2004.

[35] R. Anderson and E. W. Mayr, *A P-complete problem and approximations to it*. Stanford University, 1984.

[36] F. E. Sevilgen, S. Aluru, and N. Futamura, "Parallel algorithms for tree accumulations," *Journal of Parallel and Distributed Computing*, vol. 65, no. 1, pp. 85–93, 2005.

[37] Y. Fang, K. Yu, R. Cheng, L. V. S. Lakshmanan, and X. Lin, "Efficient algorithms for densest subgraph discovery," *PVLDB*, vol. 12, no. 11, pp. 1719–1732, 2019.

[38] L. Chang, "Efficient maximum clique computation over large sparse graphs," in *KDD*, 2019, pp. 529–538.

[39] J. Cohen, "Trusses: Cohesive subgraphs for social network analysis," *National Security Agency Technical Report*, vol. 16, pp. 3–1, 2008.

[40] L. Chang, J. X. Yu, L. Qin, X. Lin, C. Liu, and W. Liang, "Efficiently computing k-edge connected components via graph decomposition," in *SIGMOD*, 2013, pp. 205–216.

[41] J. Cheng, Y. Ke, A. W. Fu, J. X. Yu, and L. Zhu, "Finding maximal cliques in massive networks," *TODS*, vol. 36, no. 4, pp. 21:1–21:34, 2011.

[42] J. Pei, D. Jiang, and A. Zhang, "On mining cross-graph quasi-cliques," in *KDD*, 2005, pp. 228–238.

[43] A. E. Sariyüce and A. Pinar, "Fast hierarchy construction for dense subgraphs," *PVLDB*, vol. 10, no. 3, pp. 97–108, 2016.

[44] A. E. Sariyüce, C. Seshadhri, and A. Pinar, "Local algorithms for hierarchical dense subgraph discovery," *PVLDB*, vol. 12, no. 1, pp. 43–56, 2018.

[45] W. Khaouid, M. Barsky, S. Venkatesh, and A. Thomo, "K-core decomposition of large networks on a single PC," *PVLDB*, vol. 9, no. 1, pp. 13–23, 2015.

[46] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu, "Querying k-truss community in large and dynamic graphs," in *SIGMOD*, 2014, pp. 1311–1322.

[47] J. Wang and J. Cheng, "Truss decomposition in massive networks," *PVLDB*, vol. 5, no. 9, pp. 812–823, 2012.

[48] Y. Wang, X. Jian, Z. Yang, and J. Li, "Query optimal k-plex based community in graphs," *DSE*, vol. 2, no. 4, pp. 257–273, 2017.

[49] R. Zhou, C. Liu, J. X. Yu, W. Liang, B. Chen, and J. Li, "Finding maximal k-edge-connected subgraphs from a large graph," in *EDBT*, 2012, pp. 480–491.

[50] D. Wen, L. Qin, Y. Zhang, X. Lin, and J. X. Yu, "I/O efficient core graph decomposition at web scale," in *ICDE*, 2016, pp. 133–144.

[51] J. Cheng, Y. Ke, S. Chu, and M. T. Özsu, "Efficient core decomposition in massive networks," in *ICDE*. IEEE, 2011, pp. 51–62.

[52] Y. Shao, L. Chen, and B. Cui, "Efficient cohesive subgraphs detection in parallel," in *SIGMOD*, 2014, pp. 613–624.

[53] X. Huang, W. Lu, and L. V. S. Lakshmanan, "Truss decomposition of probabilistic graphs: Semantics and algorithms," in *SIGMOD*, 2016, pp. 77–90.

[54] L. Yuan, L. Qin, X. Lin, L. Chang, and W. Zhang, "I/O efficient ECC graph decomposition via graph reduction," *VLDB J.*, vol. 26, no. 2, pp. 275–300, 2017.

[55] Y. Fang, X. Huang, L. Qin, Y. Zhang, W. Zhang, R. Cheng, and X. Lin, "A survey of community search over big graphs," *VLDB J.*, vol. 29, no. 1, pp. 353–392, 2020.

[56] F. D. Malliaros, C. Giatsidis, A. N. Papadopoulos, and M. Vazirgiannis, "The core decomposition of networks: theory, algorithms and applications," *VLDB J.*, vol. 29, no. 1, pp. 61–92, 2020.

[57] V. E. Lee, N. Ruan, R. Jin, and C. Aggarwal, "A survey of algorithms for dense subgraph discovery," in *Managing and Mining Graph Data*. Springer, 2010, pp. 303–336.

[58] Y. He, K. Wang, W. Zhang, X. Lin, and Y. Zhang, "Exploring cohesive subgraphs with vertex engagement and tie strength in bipartite graphs," *Information Sciences*, vol. 572, pp. 277–296, 2021.

[59] B. Liu, L. Yuan, X. Lin, L. Qin, W. Zhang, and J. Zhou, "Efficient (α, β)-core computation: An index-based approach," in *WWW*, 2019, pp. 1130–1141.

[60] X. Jian, Y. Wang, and L. Chen, "Effective and efficient relational community detection and search in large dynamic heterogeneous information networks," *PVLDB*, vol. 13, no. 10, pp. 1723–1736, 2020.

[61] Y. Fang, Y. Yang, W. Zhang, X. Lin, and X. Cao, "Effective and efficient community search over large heterogeneous information networks," *PVLDB*, vol. 13, no. 6, pp. 854–867, 2020.

[62] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu, "The ubiquity of large graphs and surprising challenges of graph processing," *PVLDB*, vol. 11, no. 4, pp. 420–431, 2017.

[63] U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Physical review E*, vol. 76, no. 3, p. 036106, 2007.

[64] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of statistical mechanics: theory and experiment*, vol. 2008, no. 10, p. P10008, 2008.

[65] A. E. Sariyüce, B. Gedik, G. Jacques-Silva, K. Wu, and Ü. V. Çatalyürek, "Streaming algorithms for k-core decomposition," *PVLDB*, vol. 6, no. 6, pp. 433–444, 2013.

[66] F. Zhang, Y. Zhang, L. Qin, W. Zhang, and X. Lin, "When engagement meets similarity: Efficient (k, r)-core computation on social networks," *PVLDB*, vol. 10, no. 10, pp. 998–1009, 2017.

[67] K. Wang, X. Cao, X. Lin, W. Zhang, and L. Qin, "Efficient computing of radius-bounded k-cores," in *ICDE*, 2018, pp. 233–244.

[68] R. Li, J. Su, L. Qin, J. X. Yu, and Q. Dai, "Persistent community search in temporal networks," in *ICDE*, 2018, pp. 797–808.

[69] R. Li, L. Qin, F. Ye, J. X. Yu, X. Xiao, N. Xiao, and Z. Zheng, "Skyline community search in multi-valued networks," in *SIGMOD*, 2018, pp. 457–472.

[70] K. Wang, W. Zhang, X. Lin, Y. Zhang, L. Qin, and Y. Zhang, "Efficient and effective community search on large-scale bipartite graphs," in *ICDE*. IEEE, 2021, pp. 85–96.

[71] M. Kitsak, L. K. Gallos, S. Havlin, F. Liljeros, L. Muchnik, H. E. Stanley, and H. A. Makse, "Identification of influential spreaders in complex networks," *Nature physics*, vol. 6, no. 11, p. 888, 2010.

[72] F. D. Malliaros, M.-E. G. Rossi, and M. Vazirgiannis, "Locating influential nodes in complex networks," *Scientific reports*, vol. 6, p. 19307, 2016.

[73] K. Shin, T. Eliassi-Rad, and C. Faloutsos, "Corescope: Graph mining using k-core analysis - patterns, anomalies and algorithms," in *ICDM*, 2016, pp. 469–478.

[74] J. Ugander, L. Backstrom, C. Marlow, and J. Kleinberg, "Structural diversity in social contagion," *PNAS*, vol. 109, no. 16, pp. 5962–5966, 2012.

[75] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani, "Large scale networks fingerprinting and visualization using the k-core decomposition," in *NIPS*, 2005, pp. 41–50.

[76] N. Barbieri, F. Bonchi, E. Galimberti, and F. Gullo, "Efficient and effective community search," *Data mining and knowledge discovery*, vol. 29, no. 5, pp. 1406–1433, 2015.