Fast Multilayer Core Decomposition and Indexing

Dandan Liu[†], Run-An Wang[‡], Zhaonian Zou[*], Xin Huang[§]
[†‡*]Harbin Institute of Technology, China [§] Hong Kong Baptist University, China
[†]ddliu@hit.edu.cn [‡]wangrunan@stu.hit.edu.cn [*]znzou@hit.edu.cn [§]xinhuang@comp.hkbu.edu.hk

***Abstract*—The multilayer (ML) graph model provides a robust representation of multi-sourced relationships among real-world entities, laying a solid foundation for reliable knowledge discovery. ML core decomposition is a fundamental analytical tool for ML graphs. It offers valuable insights into the dense structures in ML graphs and forms the basis for many complex analysis tasks. However, existing ML core decomposition algorithms face performance issues due to unavoidably unnecessary computations and are inherently serial, unable to fully leverage the multi-core processors. In this paper, we reformulate the search space of this problem with a tree-shaped structure called MLC-tree. Based on it, we present an efficient serial ML core decomposition algorithm that achieves improved time complexity over existing solutions and the first parallel framework for this problem by exploiting the path-decomposition of the MLC-tree. Two practical optimizations are introduced to further boost the parallel efficiency. To facilitate applications built upon ML cores, we construct a compact storage and index structure for ML cores based on the MLC-tree. The usefulness of this index is showcased through two applications: ML core search and a novel weighted densest subgraph discovery problem. Extensive experiments on 9 real-world ML graphs show that our MLC-tree-based ML core decomposition algorithm achieves a speedup of up to $128\times$ over existing baselines and the parallel approach attains an additional speedup of up to $30.6\times$ using 40 cores. Moreover, the MLC-tree index can efficiently support the studied applications.**

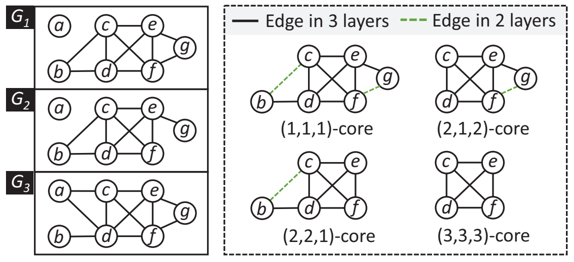
***Index Terms*—Multilayer graphs, core decomposition, parallel algorithms, the densest subgraph**

I. Introduction

Connections between real-world entities originate from various sources, such as social relationships across diverse social media platforms [1] and biomolecular relationships derived from different biological experiments [2]. **M**ulti**l**ayer (ML) graphs, structured as a series of layered graphs (layers) with each representing a specific relationship, have demonstrated the ability to provide a robust representation of such complex relationships and establish a solid foundation for reliable and accurate knowledge discovery [3]–[6].

Cohesive **s**ubgraph **m**ining (CSM), a key primitive in graph analysis, aims at finding densely connected vertices and has witnessed extensive applications [5], [7]–[11]. Among various cohesive subgraph models proposed in the literature, the notion of k-core [12] stands out for its computational efficiency [13] and hierarchical structures. The core decomposition of a graph, defined as the set of all nonempty k-cores in the graph, has proven useful in plenty of applications, such as community detection [8], anomaly detection [14], engagement dynamic modeling [15], graph visualization [10], etc.

Building upon the k-core model, Azimi et al. [16] introduced the notion of multilayer (ML) core for ML graphs.



(a) An example ML graph (b) Subgraphs induced by ML cores

Fig. 1. An example of a 3-layer ML graph and examples of ML cores.

Given an ML graph with l layers, ML cores are identified by l-dimensional non-negative integer vectors $\mathbf{k}$: the ML $\mathbf{k}$-core is the maximal subset of vertices that forms a $\mathbf{k}[i]$-core on each i-th layer. The ML core model inherits an elegant hierarchical property from the k-core: the ML $\mathbf{k}$-core is a subset of the ML $\mathbf{k}'$-core if $\mathbf{k}'$ is element-wisely no larger than $\mathbf{k}$.

Example 1. *Fig. 1(a) depicts a toy ML graph with 3 layers denoted as G_1, G_2, and G_3. These layers share the same set of vertices but have varying edge sets representing different relationships. Fig. 1(b) displays several ML cores and the subgraphs induced by them. For example, the vertex set $\{c, d, e, f\}$ forms the $(3, 3, 3)$-core as each vertex in it is adjacent to at least 3 other vertices in every layer, forming a 3-core in each layer. Moreover, the $(3, 3, 3)$-core is nested within other shown ML cores due to the hierarchical structure.*

Galimberti et al. [17] investigated the ML core decomposition problem, pioneering the study of the CSM problem in ML graphs using core-based approaches. ML core decomposition of an ML graph is defined as the set of all nonempty ML $\mathbf{k}$-cores in the graph. It provides a fine-grained view of the cohesive structures in ML graphs, revealing the connections among cohesive areas on different layers. It also serves as a foundation for a range of analysis tasks in ML graphs [18]:

1) It is proven to be an efficient pre-processing tool for solving more complex CSM problems on ML graphs such as finding cross-layer quasi-cliques [6], [19].
2) It offers a solution to the densest subgraph problem in ML graphs with a guaranteed approximation ratio.
3) It provides solutions for the community search problem in ML graphs, which identifies communities containing a given set of queried vertices.
4) Moreover, similar to the single-layer case [10], the hier-

archical structure of the ML core decomposition shows promise in guiding the visualization of ML graphs.

Computing the ML core decomposition is much more challenging than its single-layer counterpart: it has to consider each combination of layers, which results in an exponential time complexity in the number of layers. To tackle this challenge, Galimberti et al. [17], [18] use a *lattice* structure to organize all potential ML cores based on their containment relationships. They devise three algorithms differing in respective search orders on the lattice and pruning techniques to solve this problem. However, all these algorithms face performance issues. Firstly, the complex lattice structure causes all traversal orders to unavoidably visit, even compute, an ML core in the lattice multiple times, leading to massive unnecessary computations. Secondly, these algorithms heavily rely on the results of previous computations, making it challenging to parallelize and leverage the multi-core capabilities of processors.

Other researches have explored simpler models such as d-CC [5], [20] and (k, λ)-FirmCore [21], and the decomposition problems based on them. d-CC simplifies the vector $\mathbf{k}$ in the ML core model to a single integer d, and requires vertices to form d-cores on given layers. In a (k, λ)-FirmCore [21], vertices must connect to $\geq k$ other vertices in $\geq \lambda$ layers. Decomposition problems for these two models have lower time complexities of $O(2^{|L|}|\mathcal{G}|)$ and $O(|L||\mathcal{G}|)$, respectively. However, these two models impose inflexible cohesive constraints, i.e., employ the same minimum degree requirement on each layer, overlooking the inherently different cohesiveness of varying layers. This limitation makes them unsuitable as a substitute for the ML core model in many scenarios. For example, in ML social networks describing both the '*who-follows-who*' relationship on an online social platform and the '*friendship*' connections in real life, vertices in the former layer are certainly expected to have more neighbors. Using the same minimum degree constraints for both layers is evidently inappropriate and may lead to undesired results.

Given the above limitations in existing researches and the diverse applications built upon ML cores, this paper revisits the ML core decomposition problem, develops more efficient solutions, and addresses the associated ML core indexing problem that builds a compact storage and index structure for ML cores to facilitate further analytical tasks.

To efficiently solve the ML core decomposition problem, we overcome the limitations in the existing lattice-based solutions [18] by both reducing the search space and introducing parallelism. Specifically, we reformulate the search space for this problem using a simple tree-shaped structure called MLC-tree. The MLC-tree determines a DFS-order generation of the ML core decomposition, ensuring a unique visit to each ML core and improving the time complexity over lattice-based approaches. Going a step further, we explore parallel solutions. The distinct features of ML cores – notably smaller in size than the k-cores but numerous – make existing parallel core decomposition methods [22], [23] inefficient in computing ML cores. Facing this challenge, we propose a novel path-level-parallel paradigm tailored to this problem, which concurrently computes ML cores on independent paths of the MLC-tree. To further boost efficiency, we design two optimization strategies including core-level-parallel startup and path merging. To the best of our knowledge, this is the first parallel algorithm for the ML core decomposition problem in the literature.

Leveraging the containment relationships among ML cores embedded in the MLC-tree, we augment the MLC-tree and build a compact storage and index structure for ML cores. We showcase an index-based ML core search algorithm running in linear time in the height of the MLC-tree and the size of the queried ML core. As a further application, we generalize the existing densest subgraph problem [18] in ML graphs to incorporate layer weights, allowing for considering users' different preferences across layers. An efficient solution with guaranteed quality of the results is built upon the index.

We conducted extensive experiments on 9 real-world ML graphs to evaluate the performances of the proposed algorithms and highlight the following results: (1) Our MLC-tree-based ML core decomposition algorithm achieves a speedup of $3 - 128\times$ compared to the fastest lattice-based algorithms across tested graphs. (2) The parallel decomposition algorithm attains a speedup of up to $30.6\times$ on 40 cores with hyper-threading over the enhanced MLC-tree-based serial baseline. (3) The MLC-tree index demonstrates comparable performance in supporting ML core search compared with the hash-table-based naïve storage while attaining $11\% - 98\%$ space reductions. Additionally, it facilitates the efficient detection of qualified weighted dense subgraphs in ML graphs.

Contributions: The main contributions of this paper are summarized as follows:

1) A tree-shaped search space for the ML core decomposition problem and an efficient DFS-based solution.
2) The first parallel framework for computing the ML core decomposition and two practical enhancement strategies.
3) A compact storage and index structure for the ML core decomposition that supports fast ML core search.
4) A novel formulation of the weighted densest subgraph problem in ML graphs and an efficient solution based on the index with guaranteed quality of the results.
5) Extensive experimental results on 9 real-world ML graphs, demonstrating the high practical performance of the proposed algorithms.

For space limitations, we leave some proofs and experimental results in the supplementary material [24].

II. Preliminaries

In this section, we introduce some notations related to the ML graph model and ML cores. Then, the problems studied in this paper are formalized.

A. Multilayer Graphs

A *multilayer graph* (*ML graph* for short) is represented by $\mathcal{G} = (V, E, L)$, where V is a set of vertices, $L = \{1, 2, \ldots, |L|\}$ is a set of layer numbers, and $E \subseteq V \times V \times L$ is a set of edges. An edge $(u, v, i) \in E$ indicates that the vertices u and v are adjacent on layer i. In the ML graph

$\mathcal{G} = (V, E, L)$, each layer $i \in L$ has its own set of edges denoted as E_i, that is, $E_i = \{(u, v) | (u, v, i) \in E\}$. The graph on layer i is denoted as $G_i = (V, E_i)$. For any vertex $v \in V$, let $N_i(v)$ be the set of neighbors of v in G_i, and let $d_i(v) = |N_i(v)|$ be the degree of v in G_i.

Given a vertex subset $S \subseteq V$, the subgraph of G_i induced by S is $G_i[S] = (S, E_i[S])$, where $E_i[S]$ is the set of edges in E_i with both endpoints in S. Similarly, the subgraph of the ML graph $\mathcal{G}$ induced by S is $\mathcal{G}[S] = (S, E[S], L)$, where $E[S]$ is the set of edges in E with both endpoints in S. The set of neighbors and the degree of any vertex $v \in S$ in $G_i[S]$ are denoted as $N_i^S(v)$ and $d_i^S(v)$, respectively.

B. Multilayer $\mathbf{k}$-Cores

Definition 1 ([25]). *Given an ML graph $\mathcal{G} = (V, E, L)$ and an $|L|$-dimensional non-negative integer vector $\mathbf{k} = [k_i]_{i \in L}$, the multilayer $\mathbf{k}$-core (ML $\mathbf{k}$-core for short) of $\mathcal{G}$ is a maximal subset $C \subseteq V$ such that $d_i^C(v) \geq k_i$ for all $v \in V$ and $i \in L$.*

For ease of notation, we use $C_\mathbf{k}$ to denote an ML $\mathbf{k}$-core of $\mathcal{G}$, and accordingly, $\mathbf{k}$ is called the *coreness vector* of $C_\mathbf{k}$. Galimberti et al. [18] have introduced the following two elegant properties of ML cores.

Property 1 (Uniqueness). *Given an ML graph $\mathcal{G} = (V, E, L)$ and an $|L|$-dimensional integer vector $\mathbf{k} = [k_i]_{i \in L}$, there is one unique ML $\mathbf{k}$-core $C_\mathbf{k}$.*

Property 2 (Containment). *Given an ML graph $\mathcal{G} = (V, E, L)$ and two $|L|$-dimensional integer vectors $\mathbf{k} = [k_i]_{i \in L}$ and $\mathbf{k}' = [k'_i]_{i \in L}$, we have $C_\mathbf{k} \subseteq C_{\mathbf{k}'}$ if $k'_i \leq k_i$ for all $i \in L$.*

C. Problem Formulation

Aiming for fast computation of the ML core decomposition and facilitating tasks built upon ML cores, this paper addresses the following two problems:

Problem 1. ***(Multilayer Core Decomposition, ML-CD)*** *Given an ML graph $\mathcal{G} = (V, E, L)$, find the set of all nonempty ML $\mathbf{k}$-cores of $\mathcal{G}$ for all possible coreness vectors $\mathbf{k}$, which is referred to as the ML core decomposition of $\mathcal{G}$.*

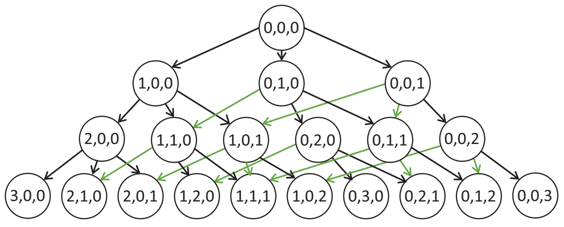
Problem 2. ***(Multilayer Core Indexing, ML-CI)*** *Given an ML graph $\mathcal{G} = (V, E, L)$, build a data structure to compactly store the ML core decomposition of $\mathcal{G}$, enabling fast retrieval of any ML $\mathbf{k}$-core given a coreness vector $\mathbf{k}$.*

III. Multilayer Core Decomposition (ML-CD)

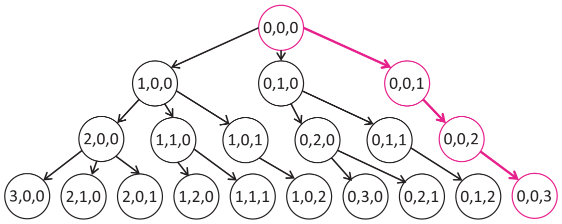
In this section, we first discuss the existing solutions [18] to the ML-CD problem and then propose novel and more efficient algorithms based on a tree-shaped search space.

A. Review of Lattice-based ML-CD Algorithms

Given an ML graph $\mathcal{G} = (V, E, L)$, the existing solutions to the ML-CD problem on $\mathcal{G}$ utilize various search strategies on the search space represented as a *core lattice* as depicted in Fig. 2(a). The core lattice is a directed acyclic graph (DAG), where each node represents a coreness vector, which uniquely corresponds to an ML core in $\mathcal{G}$ (Property 1), and each edge



(a) Core-lattice of a 3-layer graph



(b) MLC-tree of a 3-layer graph

Fig. 2. Core lattice and MLC-tree of a 3-layer graph.

represents the containment relationship between two end ML cores. We refer to the node representing the ML $\mathbf{k}$-core $C_\mathbf{k}$ as the $\mathbf{k}$-node. In the core lattice, the $[0]^{|L|}$-node is the single root node. The $\mathbf{k}$-node is the father of the $\mathbf{k}'$-node if $\mathbf{k}'$ is obtained by increasing exactly one component of $\mathbf{k}$ by 1.

Property 2 ensures that each node in the core lattice represents an ML core that is a subset of the ML cores represented by its father nodes. Taking advantage of this fact, Galimberti et al. [18] have proposed the following three traversal orders to search the core lattice in order to reduce the overheads of computing empty or redundant ML cores.

- **Breadth-First Search (BFS):** Nodes in the core lattice are traversed level by level. The ML $\mathbf{k}$-core represented by the $\mathbf{k}$-node is computed based on the intersection of ML cores represented by all father nodes of the $\mathbf{k}$-node.
- **Depth-First Search (DFS):** Nodes in the core lattice are visited along paths from internal nodes deep down to leaf nodes. On a search path, the ML $\mathbf{k}$-core is computed solely based on the ML core represented by the father node of the $\mathbf{k}$-node on the path.
- **Hybrid Search:** Combining BFS and DFS, nodes along the paths from the root to the leaf nodes, which have only one non-zero component in their associated coreness vectors, are visited in a DFS order, while the remaining nodes are visited in a BFS order.

Although the core lattice provides a complete representation of the containment relationships between ML cores, it involves a substantial number of edges, complicating the implementations of the above search orders. Specifically, all three traversal orders involve visiting and/or computing a node in the core lattice multiple times due to the presence of multiple father nodes of this node. Additionally, adopting any of the above traversal orders requires storing certain ML cores to facilitate subsequent traversal, leading to considerable space costs.

Algorithm 1 MLCD (Serial ML-CD)

```
Input: An ML graph $\mathcal{G} = (V, E, L)$
Output: The set $R$ of all nonempty ML cores in $\mathcal{G}$ with their coreness vectors
1: $R \leftarrow \{([0]^{|L|}, V)\}$
2: DfsDec($\mathcal{G}$, $[0]^{|L|}$, $R$)
3: return $R$
4: procedure DfsDec($\mathcal{G}, \mathbf{k}, R$)
5:     for $i \leftarrow |L|, |L|-1, \ldots, lnz(\mathbf{k})$ do
6:         $\mathbf{k}' \leftarrow \mathbf{k}$
7:         $\mathbf{k}'[i] \leftarrow \mathbf{k}'[i] + 1$
8:         $C \leftarrow$ Peel($\mathcal{G}, \mathbf{k}'$)
9:         if $C \neq \emptyset$ then
10:            $R \leftarrow R \cup \{(\mathbf{k}', C)\}$
11:            DfsDec($\mathcal{G}[C]$, $\mathbf{k}'$, $R$)
```

B. Tree-based ML-DC Algorithm

To overcome the limitations of the core lattice, we propose to remove certain edges from it and transform it into a tree structure, as illustrated in Fig. 2(b), which we refer to as the **m**ulti-**l**ayer **c**ore tree (MLC-tree). The MLC-tree allows for a simpler traversal order to solve the ML-CD problem without requiring additional space for storing intermediate results. Furthermore, we will later demonstrate its usefulness in supporting parallel ML-CD and indexing the ML core decomposition, which facilitates fast retrieval of ML cores.

MLC-trees: For an integer vector $\mathbf{k}$, let $lnz(\mathbf{k})$ denote the index of the last non-zero component in $\mathbf{k}$. Let the $\mathbf{k}$-node be the father of the $\mathbf{k}'$-node in the core lattice. The edge between the $\mathbf{k}$-node and the $\mathbf{k}'$-node is kept in the MLC-tree if $\mathbf{k}$ and $\mathbf{k}'$ differ in a component at an index no less than $lnz(\mathbf{k})$. In the core lattice depicted in Fig. 2(a), the green edges violate this condition and are therefore discarded to form the MLC-tree. Consequently, each node in the MLC-tree, except for the root, has exactly one father node, making it a spanning tree of the core lattice.

Algorithm: The ML-CD problem can be addressed by performing a depth-first search on the MLC-tree and computing the ML core represented by each visited node. Algorithm 1 outlines the procedure. Given an ML graph $\mathcal{G} = (V, E, L)$, a set R is used to keep all discovered nonempty ML cores along with their coreness vectors. In line 1, R is initialized to contain the $[0]^{|L|}$-core represented by the root of the MLC-tree. Obviously, the $[0]^{|L|}$-core is V. Then, the algorithm calls Procedure DfsDec to conduct the depth-first search (line 2). Finally, the results kept in R are returned in line 3.

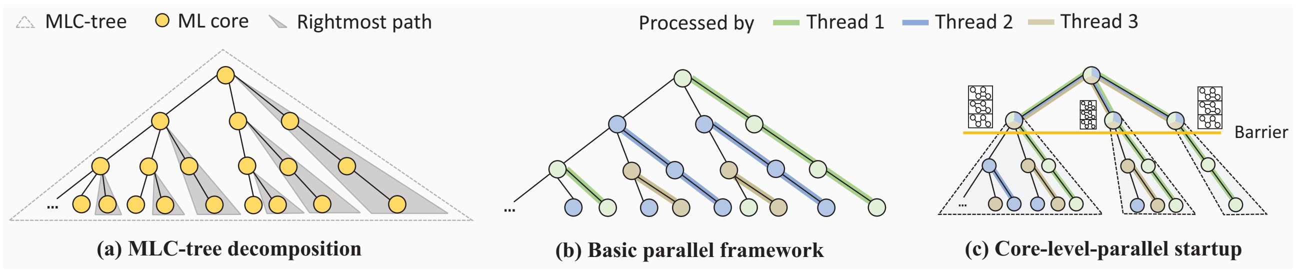
Let us delve into Procedure DfsDec. It takes 3 parameters: an ML graph $\mathcal{G}$, an integer vector $\mathbf{k}$ corresponding to the father of the node under investigation, and the result set R. The procedure performs a **for** loop (lines 5-11) that iterates over the children of the $\mathbf{k}$-node and explores the subtree rooted at each child. Inside the loop, it creates the coreness vector $\mathbf{k}'$ for a child node by incrementing a component of $\mathbf{k}$ at an index at least $lnz(\mathbf{k})$ by 1 (lines 6-7). Next, the procedure calls Function Peel to compute the ML $\mathbf{k}'$-core C on the ML graph $\mathcal{G}$ using a peeling process [18] (line 8). The peeling process iteratively removes all vertices that fail to satisfy the degree constraint imposed by $\mathbf{k}'$ on every layer. If there are vertices in C that survive the peeling process, C is added to the result set R in line 10, and the procedure continues to explore the children of the $\mathbf{k}'$-node by recursively calling Procedure DfsDec with parameter $\mathbf{k}'$ in line 11. According to Property 2, the ML core represented by any child of the $\mathbf{k}'$-node must be a subset of C, we pass the subgraph of $\mathcal{G}$ induced by C to the call to Procedure DfsDec in line 11.

Theorem 1. *Algorithm 1 finds all nonempty ML cores in the ML graph $\mathcal{G}$ in $O(\prod_{i=1}^{|L|-1} \kappa(G_i)(|L| \cdot |V| + |E|))$ time, where $\kappa(G_i)$ denotes the degeneracy of G_i.*

The proof of Theorem 1 requires the following two important concepts that will also be frequently used in the rest of the paper. For any $\mathbf{k}$-node in the MLC-tree, let $\mathbf{k}'$-node be a child of the $\mathbf{k}$-node. The $\mathbf{k}'$-node is said to be the *rightmost child* of the $\mathbf{k}$-node if $\mathbf{k}$ and $\mathbf{k}'$ are only different in their last (rightmost) components, more precisely, $\mathbf{k}'[i] = \mathbf{k}[i]$ for all $i < |L|$, and $\mathbf{k}'[|L|] = \mathbf{k}[|L|] + 1$. Furthermore, we can figure out the *rightmost path* from the $\mathbf{k}$-node to a leaf node by following the rightmost child of every encountered node until reaching a leaf node. For example, in the MLC-tree shown in Fig. 2(b), the $(0,0,1)$-node is the rightmost child of the $(0,0,0)$-node, and the rightmost path from the $(0,0,0)$-node is composed of nodes with vectors $(0,0,0)$, $(0,0,1)$, $(0,0,2)$, and $(0,0,3)$ (nodes with rose red borders).

Proof Sketch. The correctness of Algorithm 1 is guaranteed by Property 2. The DFS order enforced by Procedure DfsDec ensures that every node in the MLC-tree is visited followed by its rightmost child (if it exists). As a result, for any $\mathbf{k}$-node with $\mathbf{k}[|L|] = 0$, all ML cores represented by the nodes on the rightmost path starting from the $\mathbf{k}$-node to a leaf node are computed consecutively. This forms a complete peeling process on $\mathcal{G}[C]$, where C is the ML core represented by the father of the $\mathbf{k}$-node. This peeling process involves visiting the neighbors of all vertices in $\mathcal{G}[C]$ and removing all edges in $\mathcal{G}[C]$, which totally takes $O(|L| \cdot |C| + |E[C]|)$ time. In addition, the number of rightmost paths to be handled is equal to the number of $\mathbf{k}$-nodes in the MLC-tree with $\mathbf{k}[|L|] = 0$, which is bounded by $O(\prod_{i=1}^{|L|-1} \kappa(G_i))$. Therefore, the time complexity of Algorithm 1 is $O(\prod_{i=1}^{|L|-1} \kappa(G_i)(|L| \cdot |V| + |E|))$. □

Ordering of Layers: The order of layers in $\mathcal{G}$ determines the organization, specifically, the father-child relationships, of the nodes in the MLC-tree, thereby significantly impacting the performance of Algorithm 1. For instance, in the MLC-tree shown in Fig. 2(b), the $(1,0,0)$-node is the father of the $(1,0,1)$-node. If a graph order designates G_3 ahead of G_1, the $(0,0,1)$-node will become the father of the $(1,0,1)$-node in the new MLC-tree. Intuitively, if G_1 is denser or has a larger degeneracy than G_3, the same minimum degree constraint on G_3 holds more power to reduce the graph size than that on G_1, and thereby the $(0,0,1)$-core is likely to have a smaller size than the $(1,0,0)$-core. In this case, allowing the $(0,0,1)$-node to be the father of the $(1,0,1)$-node is more advantageous as fewer vertices need to be peeled. Therefore, we order layers in a non-decreasing order of layer density or layer degeneracy to enhance the practical efficiency of Algorithm 1.



MLC-tree
ML core
Rightmost path
Processed by
Thread 1
Thread 2
Thread 3
Barrier
(a) MLC-tree decomposition
(b) Basic parallel framework
(c) Core-level-parallel startup

Fig. 3. Illustrations of the parallel ML-CD algorithms.

C. Parallel ML-CD Algorithm

In this section, we explore parallel approaches to solving the ML-CD problem, capitalizing on the multicore capabilities of modern processors. We begin by introducing two key metrics for evaluating the performance of parallel algorithms: (1) **work**, which counts the total number of operations performed, and (2) **depth**, representing the length of the longest sequence of dependent operations. Depth quantifies the minimum running time on an infinite number of processors.

As the number of layers $|L|$ is typically small in practice, we can consider it as a constant, and the ML-CD problem is thereby polynomial-time solvable (Algorithm 1)[1]. However, as its solution encompasses the solution to the decision problem of the k-core $(k > 3)$ on every single layer, which is known to be P-complete [26], this problem is inherently sequential and is unlikely to be solved in polylogarithmic depth unless P = NC[2]. Facing this challenge, we first propose a novel MLC-tree-based parallel framework for this problem, which is then complemented by two optimizations to further boost efficiency.

1) Basic Framework: As analyzed in the proof of Theorem 1, the process of MLCD (Algorithm 1) can be seen as computing ML cores on the rightmost path starting from each $\mathbf{k}$-node in the MLC-tree, where $\mathbf{k}[|L|] = 0$, through a single peeling process. Building on this idea, we decompose the MLC-tree into a series of independent rightmost paths, as depicted in Fig. 3(a), and devise a basic parallel framework that exploits the *path-level* parallelism: the computation of ML cores following each rightmost path is carried out independently in parallel. Fig. 3(b) illustrates this idea.

Algorithm 2 outlines the implementation. It employs a similar DFS traversal on the MLC-tree as MLCD. However, it always initiates a new task for traversing each non-rightmost child of the currently visited $\mathbf{k}$-node (line 11), while the thread visiting this node proceeds to compute the ML cores represented by nodes on the rightmost path of this $\mathbf{k}$-node through a peeling process (Procedure SEARCHRMPATH). Note that during the execution, each idle thread acquires and processes a task, with no predetermined order for the task execution.

[1]When $|L|$ is large and cannot be treated as a constant, the number of ML cores can grow exponentially with $|L|$, bounded by $O(|V|^{|L|})$. It is then infeasible to solve the ML-CD problem with limited computational resources.

[2]NC is the set of decision problems decidable in polylogarithmic time on a parallel computer with a polynomial number of processors.

Algorithm 2 ParMLCD-Basic (Parallel ML-CD with the basic path-level parallelism)

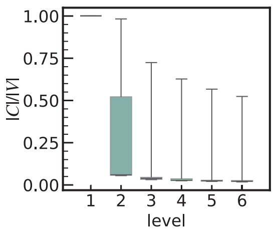
```
Input: An ML graph $\mathcal{G} = (V, E, L)$
Output: The set $R$ of all nonempty ML cores in $\mathcal{G}$ with their coreness vectors
1: $R \leftarrow \emptyset$
2: ParDfsDec($\mathcal{G}$, $[0]^{|L|}$, $R$)
3: return $R$
4: procedure ParDfsDec($\mathcal{G}$, $\mathbf{k}$, $R$)
5:     $C_{\mathbf{k}} \leftarrow$ Peel($\mathcal{G}, \mathbf{k}$)
6:     if $C_{\mathbf{k}} \neq \emptyset$ then
7:         $R \leftarrow R \cup \{(\mathbf{k}, C_{\mathbf{k}})\}$
8:         for $i \leftarrow |L|-1, |L|-2, \cdots, lnz(\mathbf{k})$ do
9:             $\mathbf{k}' \leftarrow \mathbf{k}$
10:            $\mathbf{k}'[i] \leftarrow \mathbf{k}'[i] + 1$
11:            Create a new task running ParDfsDec($\mathcal{G}$, $\mathbf{k}'$, $R$)
12:        SearchRmPath($\mathcal{G}[C_{\mathbf{k}}]$, $\mathbf{k}$, $R$)
13: procedure SearchRmPath($\mathcal{G}$, $\mathbf{k}$, $R$)
14:     while true do
15:         $\mathbf{k}[|L|] \leftarrow \mathbf{k}[|L|] + 1$
16:         $C \leftarrow$ Peel($\mathcal{G}, \mathbf{k}$)
17:         if $C \neq \emptyset$ then
18:             $R \leftarrow R \cup \{(\mathbf{k}, C)\}$
19:             $\mathcal{G} \leftarrow \mathcal{G}[C]$
20:         else break
```

Theorem 2. *Algorithm 2 finds all nonempty ML cores in the ML graph $\mathcal{G}$ in $O(\prod_{i=1}^{|L|-1} \kappa(G_i)(|L| \cdot |V| + |E|))$ work and $O(\sum_{i=1}^{|L|-1} \kappa(G_i)(|L| \cdot |V| + |E|))$ depth.*

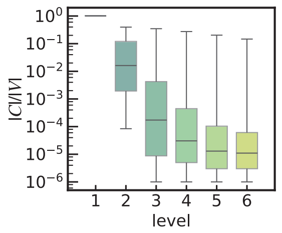
Remark: An alternative parallel framework for the ML-CD problem, inspired by its single-layer counterpart proposed in [23], is to perform the computations on each rightmost path, one after another, using available threads. By parallelizing Function PEEL and introducing the bucketing technique to dynamically maintain vertex degrees, each rightmost path can be handled in the same expected work as the serial case with a depth of $\rho|L| \cdot \log|V|$ *with high probability*[3], where ρ is bounded by the maximum number of vertices across layers. However, the practical implementation of this approach involves significant synchronization between key operators. Moreover, as ML cores typically have smaller sizes but larger numbers compared with k-cores, the computations required to obtain ML cores from their fathers cannot effectively utilize all available threads. These factors collectively result in a notably poor practical performance.

Drawbacks: This basic parallel framework has the following drawbacks: (1) Each thread needs to maintain thread-

[3]An algorithm is considered to have a $O(f(n))$ cost with high probability if it costs $O(k \cdot f(n))$ with a probability of at least $1 - 1/n^k$.



(a) Friendfeed



(b) Dblp-coauthor

Fig. 4. Sizes of the ML cores C at different levels of the MLC-tree (normalized by the total number of vertices in the input ML graph).

Algorithm 3 `P-Peel` (Parallel version of PEEL)

```
Input: An ML graph $\mathcal{G} = (V, E, L)$ and a vector $\mathbf{k} \in \mathbb{N}^{|L|}$
Output: The ML $\mathbf{k}$-core of $\mathcal{G}$
 1: Initialize global arrays $D_1, D_2, \cdots, D_{|L|}$ in parallel  ▷ $D_l[v]$ is the degree of $v$ in $G_l$ for $v \in V$ and $l \in L$
 2: Initialize global array $invalid$ in parallel ▷ $invalid[v]$ indicates whether $v$ has already been removed
 3: Initialize thread-local array $buf$ and set $s \leftarrow 0$ , $e \leftarrow 0$
 4: for $v \in V$ do in parallel
 5:     if $\exists l \in L$ s.t. $D_l[v] < \mathbf{k}[l]$ then
 6:         $buf[e] \leftarrow v$
 7:         $e \leftarrow e + 1$
    /* Each thread performs a local peeling process in lines 8–18 asynchronously.*/
 8: while $s < e$ do
 9:     $e_{pre} \leftarrow e$
10:     for $l \leftarrow 1, 2, \cdots, |L|$ do
11:         for $i \leftarrow s, s+1, \cdots, e_{pre} - 1$ do
12:             $v \leftarrow buf[i]$
13:             for $u \in N_l(v)$ do
14:                 $d_{pre} \leftarrow$ FetchAndSub$(D_l[u], 1)$
15:                 if $d_{pre} = \mathbf{k}[l]$ and $\neg$TestAndSet$(invalid[u])$ then
16:                     $buf[e] \leftarrow u$
17:                     $e \leftarrow e + 1$
18:     $s \leftarrow e_{pre}$
19: Barrier synchronization
20: return $\{v | v \in V, invalid[v] = \text{false}\}$
```

local structures for the rightmost path processing, each using $O(|L||V|)$ space, leading to huge space costs. (2) The first ML core on each rightmost path is computed from the input ML graph $\mathcal{G}$, resulting in massive repeated computations. (3) Tasks are generated interdependently. There are a limited number of tasks during the startup phase of the execution, which leads to an underutilization of available threads.

2) Core-level-parallel Startup: We introduce a *core-level* parallel paradigm in the startup phase of the decomposition to handle the limitations of the path-level parallelism. This approach is grounded in the observation that the sizes of ML cores substantially degrade as their levels in the MLC-tree become lower, which is due to the combinational degree constraints from each layer. Fig. 4 showcases the size distributions of ML cores on different levels of the MLC-tree for two real-world ML graphs Friendfeed [18] and DBLP-coauthor [27]. We can see that the median sizes of ML cores on the third level degrade to 4% and 0.02% of the number of vertices in the input graph, respectively. This suggests that a large number of vertices will be peeled when computing ML cores on upper levels, which offers the potential for efficient parallelization.

Algorithm 3 outlines `P-Peel`, a parallel version of Funtion PEEL used for computing ML cores on upper levels of the MLC-tree. It is built on two atomic instructions: `FetchAndSub` [28] and `TestAndSet` [29]. Given a memory address $addr$ and an integer δ, `FetchAndSub` performs an atomic decrement on the value at $addr$ by δ and returns the old value stored at $addr$. `TestAndSet` takes a memory address $addr$ of a boolean value as input and atomically sets the value at $addr$ to *true* and returns the old value.

`P-Peel` works as follows. It initializes global arrays D_l for $1 \leq l \leq |L|$ to track vertex degrees and array $invalid$ to mark removed vertices (lines 1–2). Each thread maintains a local array buf that functions as a queue and tracks its head and tail with variables s and e, respectively (line 3). Lines 4–7 identify the first batch of vertices to be removed, i.e., those with a degree smaller than $\mathbf{k}[l]$ in some layer l. This process is executed concurrently, and each thread adds its discovered vertices into its own queue buf. Then, in lines 8–18, each thread carries out a local peeling process asynchronously based on its queue: for every vertex v in the queue, it iterates over v's neighbors, atomically updates the degrees of the neighbors, and removes those (add those to the queue) failing to satisfy the degree constraint imposed by $\mathbf{k}$. The removal of vertices is marked atomically to ensure that each vertex is removed only once. After all threads have finished their execution, the remaining vertices collectively form the ML $\mathbf{k}$-core.

Replacing Function PEEL in line 8 of `MLCD` (Algorithm 1) with `P-Peel` establishes our core-level-parallel algorithm to compute ML cores. We adopt this approach to compute ML cores in the startup phase of the decomposition, specifically focusing on computing those on the first l levels of the MLC-tree, where l is a user-specified integer. Then, by designating the nodes on level l as new roots, we apply the path-level parallel approach (Algorithm 2) to process the subtrees rooted at them. According to Property 2, all ML-cores represented by nodes in a subtree are subsets of the one represented by the root of the subtree (*root core* for short), which generally has a much smaller size than the vertex set of the input ML graph. Therefore, we construct subgraphs induced by the root cores of these subtrees and extract the rest of the ML cores from them. Fig. 3(c) illustrates the whole process.

The above hybrid parallel framework typically outperforms the basic one in both space utilization and execution time. Firstly, it avoids the need for thread-local structures that cost $O(|L||V|)$ space. Secondly, it enables the computation of ML cores on smaller subgraphs, alleviating the extensive repeated computations involved in peeling from the input ML graph. Additionally, it increases thread utilization during the startup phase of the execution.

It should be noted that updating vertex degrees with multiple threads during the execution of `P-Peel` unavoidably causes thread contention. When computing ML cores on upper levels of the MLC-tree, which usually contain a large number of vertices, the thread contention is light. As the levels become lower, the contention intensifies. Therefore, for graphs with a large number of layers, we adopt a smaller l as the number of threads increases to mitigate the thread contention. However, for graphs with fewer layers, we opt for a larger value of l when more threads are available. The aim is to generate sufficient tasks to leverage the path-level parallelism, at the expense of slightly higher thread contention overhead.

Remark: We can develop an alternative parallel implementation of Function PEEL following the idea in [23]. Specifically, in each iteration of the peeling process, edges adjacent to all removed vertices are collected and grouped in parallel. In this way, the changes in vertex degrees can be aggregated and updated without conflicts. However, when practically applied to compute ML cores on upper levels of the MLC-tree, the overhead of heavy grouping and aggregating operations often outweighs the benefits of non-conflicting degree updates.

3) Path merging: The startup phase picks a set of nodes as new roots and builds small subgraphs induced by the ML cores represented by these new roots for computing ML cores in their respective subtrees. As the decomposition progresses to lower levels of a subtree, the sizes of ML cores generally become notably smaller in comparison to the root core of the subtree. As a result, peeling from the root core to obtain these small ML cores emerges as the most time-consuming step within the processing of rightmost paths starting from the tree nodes representing these small cores.

To tackle this issue, we implement a *path merging* strategy: when a particular ML core C is identified with a size smaller than $\alpha \cdot |C_{root}|$, where $\alpha \in [0,1]$ is a user-specified parameter, and $|C_{root}|$ is the size of the root core of the subtree, we let the thread obtaining C to continually compute all C's descendant ML cores using Algorithm 1. This strategy reduces repeated peeling operations for obtaining the descendant tiny ML cores from the root core, improving the overall efficiency.

Typically, a larger α reduces repeated computations among the processing of different rightmost paths, but it may lead to obvious load imbalance. Conversely, a smaller α involves more redundant computations but offers more tasks for parallel computation and helps distribute the workload more evenly. Therefore, we opt for a smaller α as the number of threads increases to benefit from more parallelism, and a larger α for fewer threads to take advantage of the reduced computation. Moreover, when an ML graph has a small number of layers, the number of rightmost paths in its MLC-tree is limited, and thereby a smaller α is preferred.

4) Discussions: The path-level parallel framework and two optimization strategies proposed in this section can be easily adapted to address other cohesive subgraph decomposition problems in ML graphs if the following conditions are met: (1) The search space of the problem can be expressed as a tree structure; (2) The cohesive subgraphs represented by the father nodes in the tree are supersets of those represented by their children nodes. We have outlined a detailed adapting method for solving the gCore decomposition problem [4] in general ML graphs in the supplementary material [24].

IV. MULTILAYER CORE INDEXING (ML-CI)

In a number of analytical tasks in ML graphs, e.g., trial-and-error searches for ML cores with desirable characteristics and identifying the densest subgraphs [18], there is often a need to frequently access ML cores with various coreness vectors.

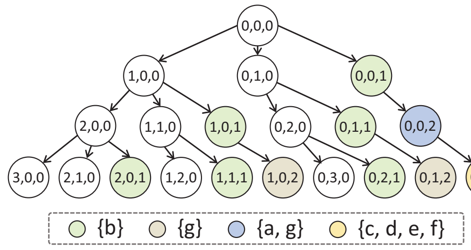


Fig. 5. Partial augmented MLC-tree for the ML graph in Fig. 1(a), with node colors representing different associated vertex sets.

One approach to realizing fast ML core retrieval involves storing the entire ML core decomposition in a hash table with the coreness vectors of the ML cores as keys. However, as mentioned in [18], the number of ML cores can grow exponentially with the number of layers, thereby making this approach impractical due to the significant storage overhead. To address this challenge, we propose a more efficient method to store and index ML cores based on the MLC-tree.

A. Storage and Index Structure

Recall that the MLC-tree systematically organizes all ML cores and preserves their partial containment relationships through the tree edges. By taking advantage of these facts, we design a storage and index structure by augmenting the MLC-tree with the following three steps:

1) Materialize all nodes in the MLC-tree that represent nonempty ML cores and the tree edges between them. We denote the materialized tree as T.
2) For each non-leaf node N in T, we associate with N the difference set between the ML core represented by N and the ML core represented by N's rightmost child.
3) For each leaf node N in T, we associate with N the ML core represented by N.

Example 2. *Fig. 5 depicts the top* 4 *levels of the augmented MLC-tree for the ML graph in Fig. 1(a). The* $\mathbf{k}$*-node* N *with* $\mathbf{k} = (0,0,2)$ *represents the ML* $\mathbf{k}$*-core* $\{a,c,d,e,f,g\}$*. The rightmost child of* N *is the* $\mathbf{k}'$*-node with* $\mathbf{k}' = (0,0,3)$*, representing the ML* $\mathbf{k}'$*-core* $\{c,d,e,f\}$*. The difference,* $\{a,g\}$*, is therefore associated with the node* N*.*

The construction of the augmented MLC-tree can be seamlessly integrated into the computation of the ML core decomposition. Whenever a node N in the MLC-tree is visited, and the ML core C represented by N is tested to be nonempty, we materialize N and associate the difference set between C and the ML core C' represented by N's father node with the father. Indeed, this difference set $C' - C$ is exactly the set of vertices peeled from C' to obtain C.

Space cost: The space cost of the augmented MLC-tree is $O(\prod_{i=1}^{|L|-1} \kappa(G_i)|V|)$ because there are $O(\prod_{i=1}^{|L|-1} \kappa(G_i))$ rightmost paths in the MLC-tree, and the difference sets stored with all nodes on each rightmost path totally contain at most $|V|$ vertices (Theorem 3, which will be introduced later). We also observe that the order of layers can affect the space cost in practice, and ordering layers in a non-decreasing order of their degeneracy often leads to a significant reduction in space cost. This is because: (1) Selecting the layer with the largest degeneracy as the $|L|$-th layer can be beneficial as it reduces the number of rightmost paths to be stored. (2) In the MLC-tree, a $\mathbf{k}$-node with $lnz(\mathbf{k}) = l$ has $|L| - l + 1$ children. Therefore, if $lnz(\mathbf{k})$ is small, it takes more space to store $|L| - l + 1$ pointers pointing to the children of the $\mathbf{k}$-node. Since the number of $\mathbf{k}$-nodes with $lnz(\mathbf{k}) = l$ is bounded by $\prod_{i=1}^{l} \kappa(G)$, this layer order can effectively save the space for storing the tree nodes.

B. Applications

Let us demonstrate how the augmented MLC-tree supports fast ML-core search and facilitates a novel dense subgraph discovery problem in ML graphs. For brevity, we will refer to the augmented MLC-tree index simply as the MLC-tree throughout the rest of this paper.

1) ML Core Search: The ML core search problem aims to find the ML core with a given coreness vector:

Problem 3. ***(Multilayer Core Search (ML-CS))*** *Given an ML graph $\mathcal{G} = (V, E, L)$ and an $|L|$-dimensional vector $\mathbf{k} = [k_i]_{i \in L}$, find the ML $\mathbf{k}$-core of $\mathcal{G}$.*

This problem is crucial for retrieving cohesive subgraphs with desirable features in ML graphs, especially in a trial-and-error manner. In addition, it serves as a useful pre-processing tool for reducing the search space of complex CSM problems in ML graphs such as the detection of (frequent) cross-layer quasi-cliques [6], [18], [19]. The following theorem lays the foundation for fast ML core search based on the MLC-tree:

Theorem 3. *For any node N in the MLC-tree, the union of vertex sets associated with nodes along the rightmost path from N to a leaf node forms the ML core represented by N.*

Theorem 3 establishes a simple MLC-tree-based ML core search approach, which is outlined in Algorithm 4. It consists of two steps: (1) Locate the $\mathbf{k}$-node N in the MLC-tree (using Procedure SEARCH in line 1); (2) Recover the ML $\mathbf{k}$-core along the rightmost path of N to a leaf node (using Procedure RECOVER in line 2). The pseudocode is straightforward, and we leave the detailed description in [24].

Theorem 4. *Algorithm 4 returns the ML $\mathbf{k}$-core C of $\mathcal{G}$ in $O(\sum_{i=1}^{|L|} \kappa(G_i) + |C|)$ time.*

2) Weighted Densest Subgraph Extraction: Existing studies have introduced ML-core-based [18] or (k, λ)-FirmCore-based [21] approximations to the densest subgraphs in ML graphs. However, in an ML graph, not all layers show equal importance to users. Some layers may be of special interest to users, and prioritizing the consideration of the cohesiveness in these layers is expected. This motivates us to study a weighted version of the densest subgraph problem:

Problem 4. ***(Weighted Densest Subgraph Discovery (WDS))*** *Given an ML graph $\mathcal{G} = (V, E, L)$, a positive real number β, $|L|$ positive real numbers $w_1, w_2, \cdots, w_{|L|}$, and a real-valued weighted density function $\rho : 2^V \to \mathbb{R}^+$ defined as*

$$\rho(S) = \max_{\hat{L} \subseteq L} \min_{i \in \hat{L}} w_i |\hat{L}|^\beta \frac{|E_i[S]|}{|S|}, \tag{1}$$

find a subset $S^* \subseteq V$ that maximizes $\rho(S^*)$.

Algorithm 4 MLCS (ML Core Search)

```
Input: The MLC-tree $T$ for an ML graph $\mathcal{G} = (V, E, L)$ and a vector $\mathbf{k} \in \mathbb{N}^{|L|}$
Output: The ML $\mathbf{k}$-core in $\mathcal{G}$
 1: $N \leftarrow$ SEARCH($T, \mathbf{k}$)
 2: return RECOVER($N$)
 3: procedure SEARCH($T, \mathbf{k}$)
 4:     $N \leftarrow$ the root of the MLC-tree $T$
 5:     $i \leftarrow 1$
 6:     while $i \le |L|$ do
 7:         $(N, i) \leftarrow$ FORWARD($N, i, \mathbf{k}$)
 8:     return $N$
 9: procedure FORWARD($N, i, \mathbf{k}$)
10:     if $\mathbf{k}_N[i] < \mathbf{k}[i]$ then        ▷ $\mathbf{k}_N$ is the coreness vector represented by $N$
11:         for each child $N'$ of $N$ do
12:             if $\mathbf{k}_N[i] \ne \mathbf{k}_{N'}[i]$ then
13:                 return $(N', i)$
14:     else
15:         return $(N, i+1)$
16: procedure RECOVER($N$)
17:     $C \leftarrow$ the vertex set associated with $N$
18:     repeat
19:         $N \leftarrow$ the rightmost child of $N$
20:         $S \leftarrow$ the vertex set associated with $N$
21:         $C \leftarrow C \cup S$
22:     until $N$ is a leaf node
23:     return $C$
```

As with the unweighted density function proposed in [18], the parameter β in Eq. (1) controls the trade-off between high density and the number of layers exhibiting that density.

By setting larger weights to the layers of users' interest, the cohesiveness of subgraphs on these layers is strengthened, while those on less relevant layers (with relatively small weights) are weakened. The unweighted densest subgraph problem in ML graphs is known to be NP-hard [18]. As the unweighted version is a special case of our WDS problem when $w_1 = w_2 = \cdots = w_{|L|}$, our WDS problem is also NP-hard, unless P = NP.

We will next establish the connections between ML cores and the weighted densest subgraph, followed by the solution for the WDS problem based on the MLC-tree.

Theorem 5. *Let $\mathcal{C}$ be the ML core decomposition of $\mathcal{G}$, C^* be the ML core in $\mathcal{G}$ that maximizes $\rho(C^*)$ (Eq. (1)), i.e., $C^* = \arg\max_{C \in \mathcal{C}} \rho(C)$, and S^* be the optimal solution to the WDS problem on $\mathcal{G}$, we have*

$$\rho(C^*) \ge \frac{w^-}{2w^+ |L|^\beta} \rho(S^*), \tag{2}$$

where $w^- = \min_{i \in L} w_i$ and $w^+ = \max_{i \in L} w_i$.

Theorem 5 provides a foundation for an efficient solution to the WDS problem based on the MLC-tree. The idea is to make a DFS traversal on the MLC-tree and identify the ML-core that maximizes $\rho(\cdot)$ as the result. Due to limited space, we leave the pseudocode of the algorithm in the supplementary material [24] and introduce some key techniques next.

The organization of the MLC-tree ensures that ML cores on each rightmost path are nested within each other. This property allows us to incrementally compute the information needed to calculate the weighted density of these ML cores, including the number of vertices and the number of edges in each layer of the subgraph induced by the ML core. Specifically, during the DFS traversal along a rightmost path, we first go straight to its leaf node and identify the densest ML core represented by the nodes on the path during the backtracking process. Let N and N' be two nodes on this rightmost path, with N being the father of N'. Suppose C and C' are the ML cores represented by N and N', respectively. As we backtrack to node N, the ML core C' and the per-layer edge numbers, say $m_1, m_2, \ldots, m_{|L|}$, in $\mathcal{G}[C']$, are already known. Let S be the vertex set associated with N. According to Theorem 3, we have $|C| = |C'| + |S|$, and the per-layer edge numbers in $\mathcal{G}[C]$ can be obtained by adding each $|\Delta_i|$ to the corresponding m_i, where Δ_i is the set of edges in the i-th layer newly introduced due to the inclusion of S, i.e., the ones with one endpoint in S and the other in $S \cup C'$.

Moreover, if we augment the MLC-tree by storing each Δ_i associated with the node N during the MLC-tree construction at a slightly higher space overhead, restoring the per-layer edge numbers can be achieved by adding each Δ_i stored in the node to m_i. We refer to this augmented MLC-tree as the *edge-difference-augmented MLC-tree*. We have compared the basic MLC-tree and the edge-difference-augmented MLC-tree by experiments. We observe that the latter shows significant efficiency improvements in supporting solving the WDS problem, with about an average of 10% increase in the space cost.

Theorem 6. *Given an ML graph $\mathcal{G}$, a $\frac{w^-}{2w^+|L|^\beta}$-approximation to the optimal solution of the WDS problem on $\mathcal{G}$ can be obtained in $O(\prod_{i=1}^{|L|-1} \kappa(G_i)(|L| \cdot |V| + |E|))$ time using the basic MLC-tree or $O(\prod_{i=1}^{|L|} \kappa(G_i))$ time using the edge-difference-augmented MLC-tree.*

Notably, when searching for the weighted densest ML cores, each rightmost path can be independently explored, enabling a path-level parallel execution, which is akin to the one employed in the parallel ML core decomposition (Algorithm 2).

V. Experiments

A. Experimental Setup

Datasets. We conducted experiments on 9 real-world ML graphs, with their characteristics presented in Table I. The first 6 graphs were obtained from [18]. Datasets DBLP-Large [27] and FlickrGrowth [30], from the KONECT Project[4], and Wiki [31], from the SNAP Datasets[5], are transformed into ML graphs by organizing their edges into different layers based on the timestamps. The graph abbreviations are shown in bold.

Environment. The experiments were conducted on a server equipped with a 40-core Intel Xeon Gold 5218R processor, supporting two-way hyper-threading, and 754GB of RAM.

[4]https://konect.cc/

[5]https://snap.stanford.edu/data/

TABLE I
Properties of Graphs Used in Experiments.

Graph	$\|V\|$	$\|E\|$	$\|L\|$	$\min_{l \in L} \|E_l\|$	$\max_{l \in L} \|E_l\|$
Sacch**C**ere	6.5k	247k	7	1.3k	91k
DBLP-**S**mall	513k	1.0M	10	96k	113k
Obamain**I**srael	2.2M	3.8M	3	557k	1.8M
Amazon	410k	8.1M	4	899k	2.4M
Higgs	456k	13M	4	28k	12M
Friend**f**eed	505k	18M	3	266k	18M
DBLP-**L**arge	1.8M	10M	22	107	2.6M
Wiki	1.0M	2.9M	10	347	988k
Flickr**G**rowth	2.3M	23M	6	121k	13M

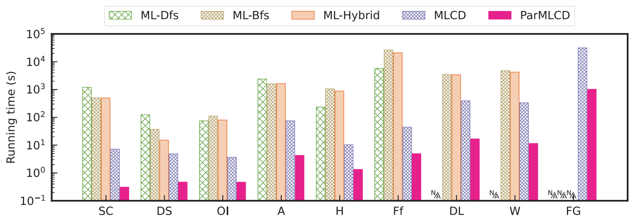


Fig. 6. Running time of ML-CD algorithms.

Source codes from [18] and [21] are compiled using Cython, while the algorithms proposed in this paper are implemented in C++ and compiled with GCC 9.4.0, both utilizing `-O3` optimization. OpenMP is used to express parallelism.

B. Performance of ML-CD.

MLC-tree V.S. Core lattice. We begin by demonstrating the effectiveness of our reduced search space MLC-tree for the ML-CD problem. Three lattice-based ML-CD algorithms [18], namely `ML-Dfs`, `ML-Bfs`, and `ML-Hybrid`, based on different search orders on the core lattice are compared with our MLC-tree-based algorithm `MLCD` (Algorithm 1) and the parallel version `ParMLCD` (Section III-C) executed with 40 threads. Fig. 6 reports the results. The empty bars labeled with 'N/A' indicate that the corresponding algorithms could not finish within 12 hours. We observe a remarkable speedup of $3 - 128\times$ with `MLCD` compared to all lattice-based algorithms across tested graphs. Moreover, the parallel approach attains an overall speedup ranging in $32 - 1606\times$. The improvement arises from the smarter reuse of previous computations during the DFS traversal on the MLC-tree and the guarantee that each ML core is visited and computed exactly once. This verifies the effectiveness of our MLC-tree-based approach.

Parallel efficiency. We then evaluate the efficiency of our parallel ML core decomposition algorithms, encompassing:

- `ParMLCD-Basic`: the basic parallel framework using path-level parallelism (Alogrithm 2).
- `ParMLCD-S`: `ParMLCD-Basic` equipped with the core-level-parallel startup (Section III-C2).
- `ParMLCD-M`: `ParMLCD-Basic` using the path merging strategy (Section III-C3).
- `ParMLCD`: `ParMLCD-Basic` with all optimizations.

We configured the parameters l and α used in the core-level-parallel startup and path merging as 3 and 0.1, respectively.

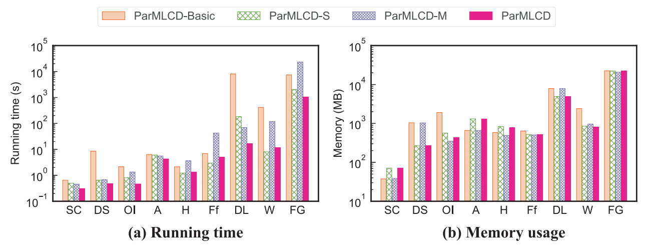


Fig. 7. Comparison between parallel ML-CD algorithms with different optimizations in terms of (a) running time and (b) memory usage.

Fig. 7 depicts the running time and memory costs of these algorithms on different graphs. Our observations include:

(1) `ParMLCD-S` exhibits significant improvements over `ParMLCD-Basic` in terms of both running time and memory usage for *sparse* graphs like DBLP-Large (DL), where the sizes of ML cores dramatically decrease as their levels in the MLC-tree increase (see Fig. 4(b)). For example, it achieves a notable speedup of $43.5\times$ and a 37% space reduction on DL. This is because the subgraphs constructed in the startup phase are much smaller than the original ML graph, enhancing the efficiency. Furthermore, it avoids the allocation of per-thread local structures with space costs proportional to the size of the input ML graph. However, for *dense* ML graphs like Amazon (A), where vertices exhibit high cohesiveness across layers, `ParMLCD-S` attains a small speedup because the sizes of ML cores decrease slowly, and the decomposition process benefits less from the constructed subgraphs. Moreover, `ParMLCD-S` may cost more space than the basic version in such graphs, as each built subgraph has a similar size to the input ML graph.

(2) The standalone adoption of the path merging strategy on `ParMLCD-Basic` demonstrates improved computational efficiency on most graphs. Nevertheless, we also observe performance degradations on certain graphs like Friendfeed (Ff). This is because the default value of $\alpha = 0.1$ is too large for such sparse ML graphs with few layers, leading to severe load imbalance. It can be alleviated by adopting a smaller α value. Besides, `ParMLCD-M` incurs basically the same space cost as `ParMLCD-Basic`. However, in scenarios where the path merging strategy substantially reduces the number of tasks, `ParMLCD-M` exhibits lower space costs.

(3) By leveraging both the core-level-parallel startup and the path merging strategies, `ParMLCD` attains a speedup ranging from $1.49\times$ to $478.8\times$ compared with `ParMLCD-Basic`, with a relative space cost between 0.23 and 1.97, depending on the graph characteristics. In fact, these two strategies complement each other: performing ML-CD on small subgraphs built in the startup phase helps alleviate the load imbalance caused by path merging, enhancing its benefits. Meanwhile, path merging reduces overall tasks, resulting in a smaller space consumption compared to `ParMLCD-S`.

Fig. 8 shows the speedup of `ParMLCD` over the serial `MLCD` (Algorithm 1) and its relative memory usage for varying numbers of threads. When using 40 threads, `ParMLCD` achieves a speedup ranging from $7.7\times$ to $30.6\times$ and incurs a relative increase in space costs from 0.02 to 3.44. It is also observed that ML graphs with a larger number of layers, such as DBLP-Large (DL), benefit more from the parallelism. The MLC-trees for such graphs hold more rightmost paths, allowing for better utilization of available threads. However, for graphs with few layers like Amazon (Fig. 8(d)) and Higgs (Fig. 8(f)), both comprising 4 layers, increasing the number of threads may lead to performance degradations. This is because when processing a node, as many new tasks as the number of the non-rightmost children of this node are created. In graphs with fewer layers, there are fewer tasks generated, leading to underutilization of threads. As a result, the increasing thread overhead outweighs the benefits of parallelism. Moreover, our core-level-parallel startup strategy helps alleviate the increase in the space occupation with the growing number of threads.

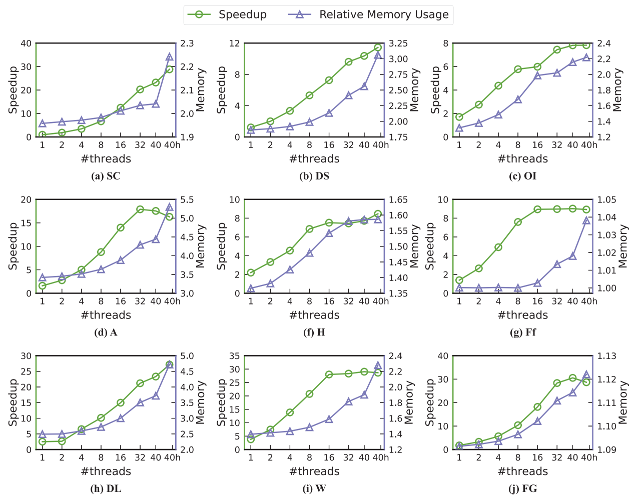


Fig. 8. The speedup and relative memory cost of `ParMLCD` compared to `MLCD`. "40h" refers to 80 hyper-threads.

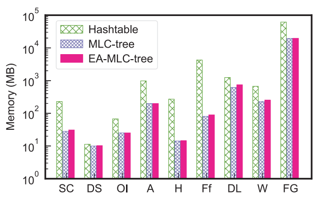


Fig. 9. Memory comparison.

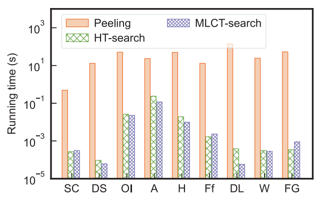


Fig. 10. Efficiency of `ML-CS`.

C. Performance of ML-CI.

We next evaluate the effectiveness of our MLC-tree index.
Space reduction. As a key metric for evaluating an index, we examine the space efficiency of our MLC-tree-based index over the naïve storage that stores each nonempty ML $\mathbf{k}$-core in a hash table with the key $\mathbf{k}$. Fig. 9 presents the results, where "Hashtable", "MLC-tree" and "EA-MLC-tree" denote the hash-table-based storage, the basic augmented MLC-tree, and the edge-difference-augmented MLC-tree used to sup-

port the densest subgraph discovery, respectively. We observe $11.2\%-98.1\%$ space reductions in the MLC-trees compared to the hash tables, highlighting the redundancy between ML cores and the promising ability of the MLC-tree to mitigate such redundancy. Moreover, we see marginal increases in space overhead for EA-MLC-trees compared to basic MLC-trees, ranging from 0.01% to 20.8% across tested graphs.

Supporting ML-CS. To evaluate the effectiveness of the MLC-tree index in supporting ML core search, we compare the running time of three ML-CS approaches:

- `Peeling`: peeling-based search that computes the ML $\mathbf{k}$-core by iteratively removing all vertices failing to satisfy the degree constraints imposed by $\mathbf{k}$.
- `MLCT-search`: MLC-tree-based search (Algorithm 4).
- `HT-search`: hash-table-based search that searches the hash table holding the ML cores with the key $\mathbf{k}$.

We run 1000 ML core search queries using these algorithms. To avoid obtaining massive empty results, we generate queried vectors $\mathbf{k}$ by randomly sampling each component $\mathbf{k}[i]$ in the range $[0, \kappa(G_i)/4]$. Fig. 10 reports the results, and our observations are as follows: (1) `MLCT-search`, based on a pre-computed MLC-tree index, significantly outperforms the peeling-based approach, achieving a speedup ratio ranging from 2 to 6 orders of magnitude across all tested graphs. (2) `MLCT-search` exhibits competitive, and in some cases superior, running time compared to the hash-table-based search. Importantly, the space overhead of the MLC-tree index is much less than that of the hash table.

Supporting WDS. We test the effectiveness of the MLC-tree index in supporting the weighted densest subgraph discovery in ML graphs. We assess the performance of the following densest subgraph detection algorithms in terms of efficiency:

- `Lattice-DS` [18]: compute the ML core decomposition using the lattice-based algorithm `ML-Bfs` and output the densest ML core as an approximation.
- `FC-DS` [21]: compute the FirmCore decomposition and output the densest FirmCore as an approximation.
- `MLCD-DS`: compute the ML core decomposition using the MLC-tree-based parallel algorithm `ParMLCD` (Section III-C) and output the densest ML core.
- `MLCT-DS`: weighted densest subgraph search algorithm using the MLC-tree index (Section IV-B2).
- `EA-MLCT-DS`: variant of `MLCT-DS` using the edge-difference-augmented MLC-tree, EA-MLC-tree for short.

As `Lattice-DS` and `FC-DS` cannot handle weighted cases, we set each $w_i = 1$ and $\beta = 2$ for the efficiency test. Fig. 11 shows the results. Note that we ran both `MLCD-DS` and `MLCT-DS` using 1 and 40 threads, with the running time range depicted by the black error bars on corresponding bars for these algorithms. We have observed that: (1) For algorithms without indexes, `MLCD-DS` consistently outperforms the lattice-based approach `Lattice-DS`, confirming the effectiveness of our tree-shaped search space. Notably, `MLCD-DS` even demonstrates better performance compared to `FC-DS` for certain graphs, despite the FirmCore decomposition problem having a linear time complexity. (2) Leveraging the pre-computed MLC-tree index, `MLCT-DS` achieves time savings of up to 88.5% compared to `MLCD-DS` when using 40 threads. Benefiting from the edge differences in the EA-MLC-tree, `EA-MLCT-DS` attains a speedup of 1 to 3 orders of magnitude over `MLCT-DS` (40 threads) and outperforms `Lattice-DS` and `FC-DS` from previous work by $4-6$ and $2-5$ orders of magnitude, respectively.

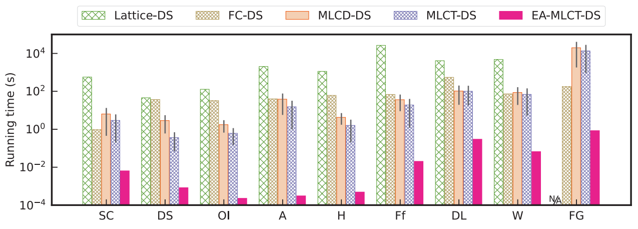


Fig. 11. Running time of the densest subgraph detection algorithms in ML graphs.

TABLE II
COMPARISONS BETWEEN APPROXIMATIONS TO THE WEIGHTED DENSEST SUBGRAPHS ON GRAPH SC

Model	l_1	l_2	l_3	l_4	l_5	l_6	l_7	ρ	$\lvert\hat{L}\rvert$
WC-App (1)	5.52	0.82	5.45	5.44	11.41	10.07	11.37	**195.81**	6
C-App	3.95	0.54	7.45	7.49	8.77	10.17	15.11	186.27	5
FC-App	3.70	12.55	5.43	0.55	11.07	11.63	27.77	177.12	4
WC-App (2)	0.44	7.01	6.94	7.02	10.27	12.60	15.02	**249.82**	6
C-App	0.40	5.42	7.45	7.49	8.77	10.17	15.11	195.19	6
FC-App	0.37	125.50	5.43	0.55	11.07	11.63	27.77	177.12	4
WC-App (3)	0.37	0.46	42.39	11.28	11.60	11.41	24.03	**282.11**	5
C-App	0.40	0.54	74.51	7.49	8.77	10.17	15.11	187.19	5
FC-App	0.37	12.55	54.30	0.55	11.07	11.63	27.77	276.75	5
WC-App (4)	0.24	0.37	2.37	83.46	26.30	4.10	26.66	**236.71**	3
C-App	0.40	0.54	7.45	74.88	8.77	10.17	15.11	186.27	5
FC-App	0.37	12.55	5.43	5.50	11.07	11.63	27.77	195.48	6
WC-App (5)	0.16	0.45	1.39	6.05	289.23	2.98	19.40	**289.23**	1
C-App	0.40	0.54	7.45	7.49	87.74	10.17	15.11	186.27	5
FC-App	0.37	12.55	5.43	0.55	110.70	11.63	27.77	186.08	4

For $i \in L$, column l_i displays the weighted density in layer i (the standard density multiplied by the layer weight). ρ is the overall weighted density defined in Eq. (1), and $|\hat{L}|$ indicates the number of layers contributing to ρ.

We proceed to examine the quality of the dense subgraphs identified by various algorithms. In this experiment, we vary the layer of priority and set its weight to 10 while keeping the others of weight 1. Additionally, β is set to 2. The results obtained on graph SacchCere (SC) are presented in Table II. Here, WC-App(x) represents the output of `EA-MLCT-DS` with the x-th layer set priority, while C-App and FC-App denote the results of `Lattice-DS` and `FC-DS`, respectively.

We can see that when varying the layer set priority, WC-App consistently shows the highest overall weighted density and guarantees a large weighted density in the prioritized layer, highlighting the effectiveness of `EA-MLCT-DS` in handling different layer preferences. Furthermore, WC-App considers a better trade-off in optimizing the density in the prioritized layer and in other layers. As we can see in cases where $x = 2$ and 3, although it displays a lower density than C-App and/or FC-App in the prioritized layer, the optimization of the density on other layers contributes to the highest overall density.

D. Impact of layer orders.

We evaluate the impact of different layer orders on the efficiency of computing the ML core decomposition and the memory usage of the MLC-tree index. The results [24] are consistent with our earlier analysis: (1) Ordering layers in non-decreasing order of their density or degeneracy generally incurs minimal time overhead for most graphs, especially for those exhibiting wide variations in layer densities and degeneracies. (2) The non-decreasing order of layer degeneracy always leads to the smallest MLC-tree.

VI. Related Work

This section presents a brief overview of the existing work related to the ML core decomposition problem.

Core Decomposition. Core decomposition is a fundamental graph-analysis tool and has seen a wide range of applications [8], [10], [14], [15], [32]. The state-of-the-art core decomposition algorithm [13] employs the vertex peeling paradigm and achieves a cost of $O(m+n)$ time, where m and n represent the number of edges and vertices in the graph, respectively. A local algorithm based on the `h-index` [33] for calculating the coreness values of vertices in a graph, equivalent to computing the core decomposition, is shown in [8]. It allows for a natural parallel implementation. Core decomposition has been extended to various types of graphs [34]–[37] and studied in distributed environment [38], [39] or on external memory [40], [41], as well as handling dynamic graphs [42], [43].

Parallel Core Decomposition. We here focus on works with shared-memory parallelism. Dasari et al. [44] and Kabir et al. [22] proposed straightforward parallelizations of the serial peeling-based core decomposition algorithm [13], with the latter featuring fewer synchronization barriers. By introducing the bucketing technique, Dhulipala et al. [23] proposed the first work-efficient parallel core decomposition paradigm. Rooted in the `h-index`-based approach, Sariyuce et al. [45] presented parallelizations in both synchronous and asynchronous manners, with the latter usually demonstrating faster convergence. Additionally, related issues, such as the dynamic maintenance of coreness values [46] and computing the core hierarchy [47], have also been investigated. However, all these parallel algorithms cannot be applied to ML graphs directly. Furthermore, the distinct features of ML cores from k-cores make the above parallel paradigms inefficient in computing ML cores.

Multilayer Core Decomposition. Azimi et al. [16] extended the k-core model to ML graphs and introduced the notion of ML $\mathbf{k}$-core. Galimberti et al. [17], [18] employed a lattice structure to organize the ML cores of an ML graph and proposed three algorithms to compute the ML core decomposition based on different search orders on the lattice and pruning techniques. However, these algorithms face scalability issues and are hard to parallelize.

Relaxations of the ML core model like the d-CC model [5], [20] and the (k, λ)-FirmCore model [21] have been studied in the literature. These models impose uniform degree constraints across all layers, simplifying the search spaces of their decomposition problems. However, these models are not suitable substitutes for the ML core model in many scenarios as their simplified constraints overlook the inherent differences in the cohesiveness of different layers.

Densest Subgraph Discovery in ML graphs. Tommaso et al. [48] surveyed researches related to the densest subgraph problem. Here, we only focus on the works studied in multilayer settings. Jethava and Beerenwinkel [49] formulated the densest common subgraph problem that finds vertex subsets maximizing the minimum density across layers and proposed both linear-programming-based and greedy-peeling-based solutions. Galimberti et al. [18] proposed a generalization of the densest common subgraph problem that exploits a trade-off between high average-degree density and the number of layers exhibiting that density. They demonstrated that the densest ML core unfolds an approximation solution to this problem. Hashemi et al. [21] presented another (k, λ)-FirmCore-based approximate solution, which improves the one given in [18] in terms of both computational efficiency and approximation guarantee. Additionally, Semertzidis et al. [50] introduced a series of variants of the densest common subgraph problem that consider different intra-layer and cross-layer information aggregation patterns, and they demonstrate the computational complexity and algorithmic solution for each case. However, the weighted densest subgraph problem explored in this paper considers users' different preferences across layers, which is not taken into account in the above works.

VII. Conclusions

Multilayer (ML) core decomposition is a fundamental tool for analyzing dense structures in ML graphs. We present a novel MLC-tree-based ML core decomposition algorithm that demonstrates improved time complexity over existing lattice-based approaches and achieves a practical speedup of up to $128\times$ on tested graphs. Based on the rightmost-path-decomposition of the MLC-tree, we propose the first parallel framework for computing the ML core decomposition. Enhanced by two optimization strategies including the core-level-parallel startup and path merging, it attains an additional speedup of up to $30.6\times$ on 40 cores over the serial version. The augmented MLC-tree offers a compact storage and index for the ML core decomposition, supporting fast retrieval of any ML core C in linear time in the height of the MLC-tree and the size of C. The further application of a novel weighted densest subgraph problem in ML graphs is also shown to be efficiently solved using the MLC-tree index, with guaranteed quality of the results. Extensive experiments verify the significant improvements in the practical performance of the proposed algorithms over existing baselines.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (Grant No. 62072138) and the Hong Kong RGC (Grant No. 22200320).

References

[1] M. E. Dickison, M. Magnani, and L. Rossi, *Multilayer social networks*. Cambridge University Press, 2016.

[2] H. Hu, X. Yan, Y. Huang, J. Han, and X. J. Zhou, "Mining coherent dense subgraphs across massive biological networks for functional discovery," *Bioinformatics*, vol. 21, no. suppl_1, pp. i213–i221, 2005.

[3] D. Luo, Y. Bian, Y. Yan, X. Liu, J. Huan, and X. Zhang, "Local community detection in multiple networks," in *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, 2020, pp. 266–274.

[4] D. Liu and Z. Zou, "gcore: Exploring cross-layer cohesiveness in multi-layer graphs," *Proceedings of the VLDB Endowment*, vol. 16, no. 11, pp. 3201–3213, 2023.

[5] R. Zhu, Z. Zou, and J. Li, "Diversified coherent core search on multi-layer graphs," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 2018, pp. 701–712.

[6] J. Pei, D. Jiang, and A. Zhang, "On mining cross-graph quasi-cliques," in *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, 2005, pp. 228–238.

[7] Z. Hammoud and F. Kramer, "Multilayer networks: aspects, implementations, and application in biomedicine," *Big Data Analytics*, vol. 5, no. 1, p. 2, 2020.

[8] L. Chang and L. Qin, "Cohesive subgraph computation over large sparse graphs," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019, pp. 2068–2071.

[9] X. Meng, H. Huo, X. Zhang, W. Wang, and J. Zhu, "A survey of personalized news recommendation," *Data Science and Engineering*, vol. 8, no. 4, pp. 396–416, 2023.

[10] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani, "k-core decomposition: A tool for the visualization of large scale networks," *arXiv preprint cs/0504107*, 2005.

[11] Y. Fang, K. Wang, X. Lin, and W. Zhang, *Cohesive Subgraph Search Over Large Heterogeneous Information Networks*. Springer, 2022.

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

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

[14] K. Shin, T. Eliassi-Rad, and C. Faloutsos, "Corescope: Graph mining using k-core analysis—patterns, anomalies and algorithms," in *2016 IEEE 16th international conference on data mining (ICDM)*. IEEE, 2016, pp. 469–478.

[15] F. D. Malliaros and M. Vazirgiannis, "To stay or not to stay: modeling engagement dynamics in social graphs," in *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, 2013, pp. 469–478.

[16] N. Azimi-Tafreshi, J. Gómez-Gardenes, and S. Dorogovtsev, "k- core percolation on multiplex networks," *Physical Review E*, vol. 90, no. 3, p. 032816, 2014.

[17] E. Galimberti, F. Bonchi, and F. Gullo, "Core decomposition and densest subgraph in multilayer networks," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, 2017, pp. 1807–1816.

[18] E. Galimberti, F. Bonchi, F. Gullo, and T. Lanciano, "Core decomposition in multilayer networks: Theory, algorithms, and applications," *ACM Trans. Knowl. Discov. Data*, vol. 14, no. 1, pp. 11:1–11:40, 2020.

[19] D. Jiang and J. Pei, "Mining frequent cross-graph quasi-cliques," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 2, no. 4, pp. 1–42, 2009.

[20] B. Liu, F. Zhang, C. Zhang, W. Zhang, and X. Lin, "Corecube: Core decomposition in multilayer graphs," in *Web Information Systems Engineering–WISE 2019: 20th International Conference, Hong Kong, China, January 19–22, 2020, Proceedings 20*. Springer, 2019, pp. 694–710.

[21] F. Hashemi, A. Behrouz, and L. V. Lakshmanan, "Firmcore decomposition of multilayer networks," in *Proceedings of the ACM Web Conference 2022*, 2022, pp. 1589–1600.

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

[23] L. Dhulipala, G. Blelloch, and J. Shun, "Julienne: A framework for parallel graph algorithms using work-efficient bucketing," in *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, 2017, pp. 293–304.

[24] (2024) Supplementary material – fast multilayer core decomposition and indexing. [Online]. Available: https://github.com/MDCGraph/MlcDec/blob/master/Appendix.pdf

[25] N. Azimi-Tafreshi, J. Gómez-Gardeñes, and S. N. Dorogovtsev, "k−core percolation on multiplex networks," *Phys. Rev. E*, vol. 90, p. 032816, Sep 2014.

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

[27] M. Ley, "The dblp computer science bibliography: Evolution, research issues, perspectives," in *International symposium on string processing and information retrieval*. Springer, 2002, pp. 1–10.

[28] __sync builtins (using the gnu compiler collection (gcc)). Accessed: 2023-11. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/__sync-Builtins.html

[29] __atomic builtins (using the gnu compiler collection (gcc)). Accessed: 2023-11. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/__atomic-Builtins.html

[30] A. Mislove, H. S. Koppula, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Growth of the flickr social network," in *Proceedings of the 1st ACM SIGCOMM Workshop on Social Networks (WOSN'08)*, August 2008.

[31] A. Paranjape, A. R. Benson, and J. Leskovec, "Motifs in temporal networks," in *Proceedings of the tenth ACM international conference on web search and data mining*, 2017, pp. 601–610.

[32] M. Charikar, "Greedy approximation algorithms for finding dense components in a graph," in *Approximation Algorithms for Combinatorial Optimization: Third International Workshop, APPROX 2000 Saarbrücken, Germany, September 5–8, 2000 Proceedings*. Springer, 2003, pp. 84–95.

[33] L. Lü, T. Zhou, Q.-M. Zhang, and H. E. Stanley, "The h-index of a network node and its relation to degree and coreness," *Nature communications*, vol. 7, no. 1, p. 10168, 2016.

[34] C. Giatsidis, D. M. Thilikos, and M. Vazirgiannis, "D-cores: measuring collaboration of directed graphs based on degeneracy," *Knowledge and information systems*, vol. 35, no. 2, pp. 311–343, 2013.

[35] F. Bonchi, F. Gullo, A. Kaltenbrunner, and Y. Volkovich, "Core decomposition of uncertain graphs," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014, pp. 1316–1325.

[36] Y. Peng, Y. Zhang, W. Zhang, X. Lin, and L. Qin, "Efficient probabilistic k-core computation on uncertain graphs," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 2018, pp. 1192–1203.

[37] C. Giatsidis, D. M. Thilikos, and M. Vazirgiannis, "Evaluating cooperation in communities with the k-core structure," in *2011 International conference on advances in social networks analysis and mining*. IEEE, 2011, pp. 87–93.

[38] A. Montresor, F. De Pellegrini, and D. Miorandi, "Distributed k-core decomposition," in *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on principles of distributed computing*, 2011, pp. 207–208.

[39] M. Ghaffari, S. Lattanzi, and S. Mitrović, "Improved parallel algorithms for density-based network clustering," in *International Conference on Machine Learning*. PMLR, 2019, pp. 2201–2210.

[40] J. Cheng, Y. Ke, S. Chu, and M. T. Özsu, "Efficient core decomposition in massive networks," in *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 2011, pp. 51–62.

[41] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo, "K-core decomposition of large networks on a single pc," *Proceedings of the VLDB Endowment*, vol. 9, no. 1, pp. 13–23, 2015.

[42] Y. Zhang, J. X. Yu, Y. Zhang, and L. Qin, "A fast order-based approach for core maintenance," in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 2017, pp. 337–348.

[43] Q. C. Liu, J. Shi, S. Yu, L. Dhulipala, and J. Shun, "Parallel batch-dynamic algorithms for k-core decomposition and related graph problems," in *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*, 2022, pp. 191–204.

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

[45] A. E. Sariyüce, C. Seshadhri, and A. Pinar, "Local algorithms for hierarchical dense subgraph discovery," *Proceedings of the VLDB Endowment*, vol. 12, no. 1, pp. 43–56, 2018.

[46] Q. C. Liu, J. Shi, S. Yu, L. Dhulipala, and J. Shun, “Parallel batch-dynamic k-core decomposition,” *arXiv e-prints*, pp. arXiv–2106, 2021.

[47] D. Chu, F. Zhang, W. Zhang, X. Lin, and Y. Zhang, “Hierarchical core decomposition in parallel: From construction to subgraph search,” in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2022, pp. 1138–1151.

[48] T. Lanciano, A. Miyauchi, A. Fazzone, and F. Bonchi, “A survey on the densest subgraph problem and its variants,” *arXiv preprint arXiv:2303.14467*, 2023.

[49] V. Jethava and N. Beerenwinkel, “Finding dense subgraphs in relational graphs,” in *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2015, Porto, Portugal, September 7-11, 2015, Proceedings, Part II 15*. Springer, 2015, pp. 641–654.

[50] K. Semertzidis, E. Pitoura, E. Terzi, and P. Tsaparas, “Finding lasting dense subgraphs,” *Data Mining and Knowledge Discovery*, vol. 33, pp. 1417–1445, 2019.