

Resource-Efficient FirmCore Decomposition on Billion-scale Multilayer Graphs

Cheng Huang
Aarhus University
cheng@cs.au.dk

Davide Mottin
Aarhus University
davide@cs.au.dk

Ira Assent
Aarhus University
ira@cs.au.dk

ABSTRACT

Multilayer (ML) graphs offer a convenient paradigm for modeling complex node-to-node interactions, such as social or semantic connections, as layers of a graph. In such graphs, *FirmCore decomposition* represents an established technique to identify cohesive groups of nodes with strong ties across layers. Unfortunately, the fastest FirmCore decomposition method fails to fully harness the resources, leading to underutilized and idle threads. Our main observation is that FirmCores enjoy a grid structure we call FC-Grid, which we exploit to distribute work among threads. Building on this structure, we introduce serial and parallel algorithms for multi-core CPUs, as well as the first GPU-based algorithm. Owing to this new design, our solutions show greatly improved performance and resource utilization. Our experiments on 12 datasets show 9 $\times$ speedup on average for our serial version FC-Grid compared to existing serial methods. Furthermore, our parallel algorithm achieves an average 100.3 $\times$ speedup over the state-of-the-art parallel algorithm. For the challenging NP-hard densest subgraph mining problem in ML graphs, our algorithms achieve 15 $\times$ speedup on average.

PVLDB Reference Format:

Cheng Huang, Davide Mottin, and Ira Assent. Resource-Efficient FirmCore Decomposition on Billion-scale Multilayer Graphs. PVLDB, 19(11): 3483 - 3496, 2026.
doi:10.14778/3836663.3836703

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://github.com/kouteisang/ParGridCPU>.

1 INTRODUCTION

Multilayer (ML) graphs offer a convenient abstraction to represent different relationships among entities, such as following others on social media [19], user interests [11], or spreading rumors [16], as different layers of a graph. The utility of ML graphs has been widely recognized across various domains, including biological networks [31, 46], social networks [14], and signal processing [64].

An effective and widely spread technique for discovering cohesive groups in graphs is the *k-core decomposition* [42], which identifies maximal subgraphs where each node has degree at least k . Owing to its simplicity, efficiency, and ability to capture various notions of graph density [21], *k-core decomposition* has been

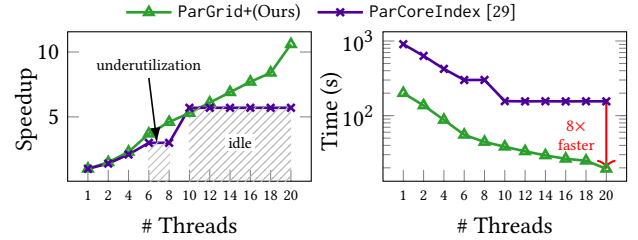


Figure 1: Per-algorithm relative speedup and runtime in number of threads for our parallel algorithm ParGrid+ and state-of-the-art parallel ParCoreIndex [29]. ParCoreIndex exhibits underutilized and idle threads, we exploit resources for greatly improved efficiency. (HW1 dataset [7, 8]).

applied to diverse domains, including detecting stock movements in financial markets [20] and biological network analysis [22]. A straightforward extension of *k-core decomposition* to multilayer (ML) graphs [23] enforces degree constraints independently on each layer. However, the resulting problem takes exponential time in the number of layers [23], making it impractical for large-scale datasets. Moreover, selecting a fixed per-layer constraint is often unrealistic for real-world ML graphs, which typically exhibit varying levels of noise [29]. To address these issues, FirmCore [29] relaxes the constraint of *k-core* on ML graphs and allows vertices to have at least k neighbors in at least λ layers.

EXAMPLE 1 (FIRMCore). Figure 2(a) shows a 3-layer social network ML graph with layers G_1 (Facebook), G_2 (Instagram), and G_3 (LinkedIn). Figure 2(b) illustrates the (2,1)-, (2,2)-, and (3,2)-FirmCore of this ML graph. For instance, vertices 0, 1, 3, and 4 form the (3,2)-FirmCore, each having at least 3 neighbors in at least 2 layers.

FirmCore decomposition [29] aims to compute the cores for all k and λ value combinations. As a result, unlike *k-core decomposition*, FirmCore decomposition achieves polynomial-time complexity and greater resilience to noise in ML graphs, making it a more palatable alternative for analyzing large real-world ML graphs. FirmCore decomposition is a fundamental primitive in a broad range of applications, including identifying cross-community collaborations in co-authorship networks, detecting user groups across social networks [29], approximating the densest subgraph [38, 49, 66], and analyzing temporal graphs [61]. These applications benefit from faster responses, as analysts may repeatedly compute and visualize [48] FirmCores to inspect cohesive structures across layers. This need becomes prominent in evolving graphs, where updates on edges may require re-running the analysis. As such, reducing the latency of FirmCore decomposition is critical for improving the responsiveness of the overall analysis workflow. Yet, the best

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.
doi:10.14778/3836663.3836703

parallel algorithm requires 15 minutes to run on large real-world graphs [27], which limits its applicability at scale. Besides these applications, FirmCore decomposition has inspired several follow-up models, such as *Generalized FirmCore* [6] and *FocusCore* [59].

To better understand the inefficiency of current algorithms for FirmCore decomposition, we observe that they suffer from structural limitations that cause poor resource utilization. The state-of-the-art serial algorithm [29] organizes FirmCores in a one-dimensional list, overlooking the inherent *two-dimensional* (k, λ) hierarchy of the problem. Likewise, the best parallel algorithm [29] suffers from thread underutilization as it can only process each list in parallel, leaving many threads idle. As a result, its speedup quickly saturates, with no further performance gains for additional threads, as seen in Figure 1 (left). Hence, existing approaches waste available hardware resources and suffer from limited scalability.

Motivation. *Can we design efficient and scalable parallel algorithms for FirmCore decomposition that improve hardware utilization?*

This motivates us to revisit FirmCore decomposition from a new structural perspective by introducing the FC-Grid grid data structure, built upon the two-dimensional (k, λ) hierarchy, and to design new ParGrid algorithms that effectively distribute work across threads by exploiting this structure. To further improve thread utilization and efficiency, we propose to compute multiple FirmCores concurrently, utilizing vertex-level parallelism within each FirmCore, enabling both inter-FirmCore and intra-FirmCore parallelism. As a result, our solutions enjoy better resource utilization and much faster runtime, as seen in Figure 1 (right; log y-axis). We further introduce the first GPU-based algorithm, ParGrid++, which processes the neighbors of each vertex in parallel to enhance the parallelism of the CPU version. To address the limited per-thread memory on GPUs, we design a novel memory-block scheme and a hierarchical parallelism strategy. Together, these enable both inter- and intra-vertex parallelization, substantially improving the overall efficiency of FirmCore decomposition. We conduct extensive experiments on 14 datasets. Results show that our algorithms significantly outperform the state-of-the-art with an average of 100.3 $\times$ speedup and 2.7 $\times$ less memory consumption. To show the practical benefit of our algorithms, we apply FC-Grid to the NP-hard problem of densest subgraph mining in ML graphs; the results show 15.3 $\times$ speedup on average compared to the state-of-the-art parallel algorithm. Our main contributions include:

- A novel two-dimensional structure, FC-Grid with two traversal strategies that enable efficient FirmCore decomposition and retrieval of any (k, λ) -FirmCore (Section 4).
- Parallel CPU algorithms leveraging vertex-level parallelism to enhance thread utilization for efficient FirmCore decomposition (Section 5.1–5.2).
- The first GPU-based FirmCore decomposition algorithm with hierarchical parallelism strategy (Section 5.3).
- Tests across 14 large-scale datasets show our serial algorithm averages 9 $\times$ speedup and 2.7 $\times$ less memory than the state-of-the-art, while our parallel versions achieve 10.3 $\times$ (CPU) and 100.3 $\times$ (GPU) speedups over existing parallel solutions (Section 6).
- We demonstrate our solutions in the real-world applications of FirmCore retrieval, ML temporal social network analysis, and densest subgraph mining. (Section 7).

2 RELATED WORK

Cohesive subgraph mining in a single-layer graph. Identification of cohesive subgraphs is a key task in graph mining with a wide range of applications [20, 22, 26]. In this category, k -core decomposition [5] is an appealing choice due to its linear time complexity. The k -core decomposition problem is to identify a maximal subgraph where each node has degree at least k . This problem has various applications, including social network analysis [50], data visualization [3], anomaly detection [58], biology [62], and graph mining [57]. Due to its broad range of applications, k -core decomposition has been extensively studied in various contexts, including parallel [1, 15, 35, 47], distributed [2, 52, 54], streaming [55, 65], and for different types of graphs [9, 25, 32, 63].

Cohesive subgraph mining in ML graphs. A ML graph is a popular representation of graphs with different types of entities and relationships. k -core decomposition in ML graphs [4] aim to identify all non-empty k -cores; given a vector $\mathbf{k}$ with length equal to the number of layers, where each entry k_l is a non-negative integer, the k -core is the maximum subgraph wherein each vertex has at least k_l neighbors for each layer $l \in \{1, \dots, |L|\}$, where $|L|$ is the number of layers in the ML graph. Galimberti et al. [23] utilize the hierarchy structure of k -cores of an ML graph and propose three algorithms for computing the ML k -core decomposition by using different traversal orders: ML-BFS (Breadth-First Search), ML-DFS (Depth-First Search), and ML-Hybrid, which combines both BFS and DFS strategies. ParMLCD [44] organizes ML k -cores into an MLC-tree structure to reduce the search space of [23], and parallelizes the construction of the MLC-tree along its rightmost path.

Dense subgraph in ML graphs. While there are various algorithms for extracting dense subgraphs in single-layer graphs [12, 36], little work explores the problem in ML graphs. The *densest common subgraph problem* in ML graphs finds the subgraph with the highest minimum density across layers [34]. ML density is a distinct notion of density [23] that balances vertex density with the number of layers, thereby enabling efficient approximate algorithms.

FirmCore Decomposition. The state-of-the-art serial algorithm CoreIndex [29] organizes the FirmCore into a *one-dimensional list* structure for each layer constraint λ , and computes the largest k such that a vertex belongs to the (k, λ) -FirmCore. Yet, this design has a structural limitation: it overlooks the natural *two-dimensional hierarchy* among FirmCores and fails to capture the dependencies and relationships across different lists. In particular, the state-of-the-art parallel algorithm ParCoreIndex [29] assigns a thread to each individual list, limiting the parallelism to the number of layers in the graph. This design naturally leads to suboptimal thread utilization and leaves some threads idle, as the number of threads often exceeds the number of layers. To address the limitations of the current solutions and accelerate FirmCore decomposition, we design the first parallel framework on both CPUs and GPUs.

3 FIRM CORE DECOMPOSITION

A *multilayer graph* (ML graph) is a triple $G = (V, E, L)$, where V is a set of vertices, $L = \{1, 2, \dots, |L|\}$ a set of layers, and $E \subseteq V \times V \times L$ a set of edges. $|V|$, $|E|$, and $|L|$ denote the number of vertices, edges, and layers, respectively. An edge $(u, v, l) \in E$ is a connection between vertices u, v in layer l . In an ML graph, each layer has its own set

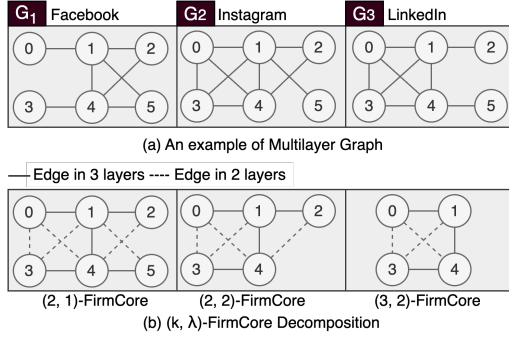


Figure 2: Example (a) 3-layer graph with each layer representing a different network, (b) Example of (k, λ) -FirmCore.

of edges $E_l = \{(u, v) | (u, v, l) \in E\}$ and its own set of vertices V_l . All layers share the same vertex set. The graph in layer l is $G_l = (V_l, E_l)$.

A neighbour in layer l of vertex v is denoted as $N_l(v)$ and the degree of v in layer l is $deg_l(v) = |N_l(v)|$. The subgraph G induced by a set of vertices $SV \subseteq V$ in layer l is the graph $G_l[SV] = (SV, E_l[SV])$ having the edges between the vertices in SV . The degree $deg_l^{SV}(v)$ in the subgraph $G_l[SV]$ is the number of neighbors of v in layer l .

DEFINITION 1 (FIRMCore). For undirected ML graph $G = (V, E, L)$, positive natural numbers λ , $1 \leq \lambda \leq |L|$, $k \geq 1$, the (k, λ) -FirmCore of G is a subgraph $G[SV]$ such that (a) for each vertex $v \in SV$ there are at least λ layers such that $deg_l^{SV}(v) \geq k$ and (b) $G[SV]$ is maximal, so there is no larger $SV' \subseteq V$, $SV \subseteq SV'$ for which (a) holds.

Hashemi et al. [29] note the following properties:

PROPERTY 1 (UNIQUENESS). Given an ML graph $G = (V, E, L)$ and thresholds k and λ , there is one unique (k, λ) -FirmCore.

PROPERTY 2 (HIERARCHICAL STRUCTURE). Given an ML graph $G = (V, E, L)$ and thresholds k and λ , the $(k, \lambda + 1)$ -FirmCore and the $(k + 1, \lambda)$ -FirmCore are subgraphs of the (k, λ) -FirmCore.

As shown in Fig. 2(b), the size of the FirmCore remains the same or decreases as k or λ increases. For example, the $(3, 2)$ -FirmCore is a subset of the $(2, 2)$ -FirmCore, and the $(2, 2)$ -FirmCore is a subset of the $(2, 1)$ -FirmCore.

We aim to efficiently compute and retrieve **all** FirmCores of an ML graph G . We formally define the problem below.

PROBLEM 1 (FIRMCore DECOMPOSITION [29]). For ML graph G , find all (k, λ) -FirmCore for **all** combinations of $k \geq 1$, $1 \leq \lambda \leq |L|$.

4 EFFECTIVE HIERARCHICAL TRAVERSAL

Here, we introduce a novel two-dimensional grid structure, FC-Grid, which organizes all (k, λ) -FirmCores into a hierarchical structure across the k and λ dimensions. FC-Grid reveals the full structural hierarchy and enables more efficient decomposition by reducing redundant computations. We observe that it naturally supports different traversal orders along both dimensions. As we will show, we can take advantage of this flexibility by selecting traversal strategies LP-Enum and DP-Enum based on the graph's properties. In contrast, previous approaches [29] employ a list-based representation of FirmCores that overlooks the structural interdependencies among

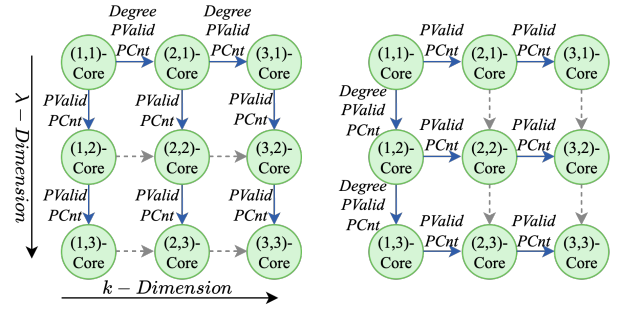


Figure 3: FC-Grid with DP-Enum (left) and LP-Enum (right).

FirmCores with different (k, λ) values. Based on FC-Grid, we propose a bucket-free serial algorithm that efficiently computes all FirmCores while no existing work has exploited the inherent two-dimensional hierarchy among (k, λ) -FirmCore for computation.

DEFINITION 2 (FC-Grid). Given a multilayer graph $G = (V, E, L)$, the FC-Grid is a two-dimensional structure that organizes all (k, λ) -FirmCores by their degree constraint k and layer constraint λ . Each cell in the grid corresponds to the (k, λ) -FirmCore.

Unlike list-based representations that support only a fixed traversal order, the two-dimensional grid structure of FC-Grid enables flexible traversal along both the k - and λ -dimensions, where the k -dimension captures degree constraint and the λ -dimension captures layer constraint.

DEFINITION 3 (DP-Enum TRAVERSAL). Given a multilayer graph $G = (V, E, L)$, DP-Enum (Degree-Prioritized Enumeration) traverses G along the k -dimension. If the $(k, 1)$ -FirmCore is non-empty, it proceeds to traverse along the corresponding λ -dimension for that k value.

DEFINITION 4 (LP-Enum TRAVERSAL). Given a multilayer graph $G = (V, E, L)$, the LP-Enum (Layer-Prioritized Enumeration) traverses G along the λ -dimension. If the $(1, \lambda)$ -FirmCore is non-empty, it proceeds to traverse along the corresponding k -dimension for that λ value.

EXAMPLE 2. Figure 3 illustrates FC-Grid with the two traversal strategies: DP-Enum (left) and LP-Enum (right). In DP-Enum, the traversal proceeds along the k -dimension. For each k , if the $(k, 1)$ -FirmCore is non-empty, it continues to increase λ within the same k until the corresponding (k, λ) -FirmCore becomes empty. Then, it increments k by one and starts from the $(k, 1)$ -FirmCore that is derived from the previous $(k-1, 1)$ -FirmCore.

Our FC-Grid reduces the computational effort compared to the state-of-the-art CoreIndex [29], pruning along the k -dimension in the DP-Enum traversal or the λ -dimension in the LP-Enum traversal. As such, FC-Grid fully harnesses Property 2, whereby (k, λ) -FirmCore is always a subset of the $(k-1, \lambda)$ -FirmCore and $(k, \lambda-1)$ -FirmCore. In contrast, CoreIndex computes each λ -list from scratch on the ML graph. We also observe that CoreIndex [29] maintains bucket structures to store top- λ degree ordering to compute each FirmCore, but maintaining such buckets incurs both time and memory overhead. More importantly, bucket-based designs are not parallel-friendly as they introduce synchronization and thread

contention due to frequent vertex movements across buckets. To address these challenges, we propose a bucket-free algorithm for constructing the grid structure, which serves as the cornerstone for our parallel algorithm design.

Algorithm 1 provides the serial algorithm (SerGrid) for building FC-Grid with LP-Enum traversal strategy; the procedure for DP-Enum follows a similar approach. We first initialize the *Count* array with all the elements set to 0. For a given degree constraint k and layer constraint λ , the *Count* array stores the number of layers in which each vertex has degree at least k (Line 1). We then compute the *Count* values for all vertices with indices greater than or equal to pos_e . If a vertex v fails to satisfy the (k, λ) -constraint, we swap it with the vertex at position pos_e and increment pos_e by 1. As a result, all vertices with indices in the range $[pos_s, pos_e]$ do not satisfy the (k, λ) -constraint (Line 3-8). Then, for each vertex v in the range $[pos_s, pos_e]$, we remove its connections to any vertex u whose index is equal to or greater than pos_e . After each removal, if vertex u no longer satisfies the (k, λ) -constraint, we repeat the process: we swap u with the vertex at position pos_e and increment pos_e by 1 (Lines 9–16). To verify if a vertex still satisfies the (k, λ) -constraint after degree updates, we exploit our *Count* structure to avoid scanning the entire ML graph when a vertex degree changes. We lazily update a vertex's *Count* only when its degree in a layer drops to exactly $k-1$, as this is the critical point where the vertex may no longer satisfy the degree's layer constraint (Line 22-23). Finally, we store all vertices with indices greater than or equal to pos_e in FC-Grid's corresponding (k, λ) -FirmCore, FC-Grid $_{k,\lambda}$, as those vertices satisfy the (k, λ) -constraint. To construct the next FirmCore, we increment the degree constraint k and build $(k+1, \lambda)$ -FirmCore directly from its parent (k, λ) -FirmCore (Line 17-19). Moreover, if the degree constraint $k=1$, we increase the layer constraint λ by 1 to construct $(1, \lambda+1)$ -FirmCore from its parent $(1, \lambda)$ -FirmCore (Line 20-21). By contrast, CoreIndex computes each list from the entire ML graph while our algorithm computes each FirmCore from its parent FirmCore, significantly reducing the computational space. Furthermore, since FC-Grid explicitly stores the valid vertex set for each FirmCore, it allows efficient retrieval of any (k, λ) -FirmCore without any preprocessing.

We define $k_{\max}$ and $\lambda_{\max}$ to facilitate the theoretical analysis of Algorithm 1.

DEFINITION 5 ($k_{\max}$ AND $\lambda_{\max}$). $k_{\max}$ is the maximum value of k such that the $(k, 1)$ -FirmCore is non-empty. Similarly, $\lambda_{\max}$ is the maximum value of λ such that the $(1, \lambda)$ -FirmCore is non-empty.

THEOREM 1. For an ML graph $G = (V, E, L)$, Algorithm 1 takes at most $O(|L| \cdot |V| \cdot \sum_{\lambda=1}^{\lambda_{\max}} \kappa(\lambda) + |L| \cdot |E|)$ time, where $\kappa(\lambda)$ is the largest degree constraint such that $(\kappa(\lambda), \lambda)$ -FirmCore is non-empty, and $\sum_{\lambda=1}^{\lambda_{\max}} \kappa(\lambda)$ is the total number of FirmCores in the ML graph.

PROOF. Algorithm 1 computes the FirmCores for each λ from 1 to $\lambda_{\max}$, where $\lambda_{\max} \leq |L|$. Let $|V_{k,\lambda}| \leq |V|$ and $|E_{k,\lambda}| \leq |E|$ denote the number of vertices and edges in the (k, λ) -FirmCore. Let $\kappa(\lambda)$ be the largest value such that the $(\kappa(\lambda), \lambda)$ -FirmCore is non-empty.

For a fixed λ , the algorithm performs $\kappa(\lambda)$ peeling iterations. In each iteration, it scans the vertex set of its parent FirmCore, resulting in a total vertex-scanning cost of $O(\sum_{k=1}^{\kappa(\lambda)} |L| \cdot |V_{k-1,\lambda}|)$. Moreover, for a fixed λ , each edge is peeled at most once, so the total

Algorithm 1: SerGrid ($k, \lambda, \deg, pos_e$)

Data: $k=1, \lambda=1, \deg, pos_e=0$; **Result:** FC-Grid

```

1 Count[·]  $\leftarrow 0_{|V|}$  ▷ Initialize Count with all elements be 0
2  $pos_s \leftarrow pos_e$ 
3 for  $id = pos_e$  to  $|V|$  do
4    $v \leftarrow V[id], cnt \leftarrow 0$ 
5   for  $l = 0$  to  $G.layer()$  do
6      $cnt \leftarrow cnt + \deg_l(v) \geq k$ 
7   Count[ $v$ ]  $\leftarrow cnt$ 
8   if  $cnt < \lambda$  then swap( $V[pos_e++], v$ )
9 while  $pos_s < pos_e$  do
10   $v \leftarrow V[pos_s++]$ 
11  for  $l = 0$  to  $G.layer()$  do
12    for  $u \in N_l(v)$  do
13      if  $index(u) \geq pos_e$  then
14         $\deg_l(u) \leftarrow \deg_l(u) - 1$ 
15        if  $\deg_l(u) = k - 1$  and  $Verify(Count, u, \lambda)$  then
16          swap( $V[pos_e++], u$ )
17 if  $pos_e < |V|$  then
18   FC-Grid $_{k,\lambda} \leftarrow V[pos_e:]$  ▷ Store the valid vertex
19    $k \leftarrow k + 1$ , SerGrid( $k, \lambda, \deg, pos_e$ )
20 if  $pos_e < |V|$  and  $k = 1$  then
21    $\lambda \leftarrow \lambda + 1$ , SerGrid( $k, \lambda, \deg, pos_e$ )
22 Function  $Verify(Count, u, \lambda)$ :
23    $Count_u \leftarrow Count_u - 1$ ; return  $Count_u = \lambda - 1$ 
```

number of edge removals is bounded by $|E_{1,\lambda-1}|$. Therefore, the time complexity for a fixed λ is $O(\sum_{k=1}^{\kappa(\lambda)} (|L| \cdot |V_{k-1,\lambda}| + |E_{1,\lambda-1}|)$.

Summing over all $\lambda=1, \dots, \lambda_{\max}$, the total running time of Algorithm 1 is $O(\sum_{\lambda=1}^{\lambda_{\max}} (\sum_{k=1}^{\kappa(\lambda)} (|L| \cdot |V_{k-1,\lambda}| + |E_{1,\lambda-1}|)))$. In the worst case, since $|V_{k-1,\lambda}| \leq |V|$, $|E_{1,\lambda-1}| \leq |E|$ and $\lambda_{\max} \leq |L|$, the worst time complexity becomes $O(|L| \cdot |V| \cdot \sum_{\lambda=1}^{\lambda_{\max}} \kappa(\lambda) + |L| \cdot |E|)$ □

Compared with the state-of-the-art algorithm CoreIndex, which has a time complexity of $O(|E| \cdot |L|^2 + |V| \cdot |L| \cdot \log |L|)$ [29], SerGrid improves the edge-related complexity from $O(|E| \cdot |L|^2)$ to $O(|E| \cdot |L|)$ in the worst case. As a result, SerGrid is beneficial for dense multilayer graphs. The vertex-related complexity of CoreIndex is $O(|V| \cdot |L| \cdot \log |L|)$, while that of SerGrid is $O(|V| \cdot |L| \cdot \sum_{\lambda=1}^{\lambda_{\max}} \kappa(\lambda))$. The term $\sum_{\lambda=1}^{\lambda_{\max}} \kappa(\lambda)$ corresponds to the total number of non-empty (k, λ) -FirmCores in the ML graph. In ML graphs, $\kappa(\lambda)$ typically decreases rapidly as λ increases, since most (k, λ) -FirmCores become empty for large values of k and λ . Therefore, although the worst-case vertex-related complexity of SerGrid may be larger, $\sum_{\lambda=1}^{\lambda_{\max}} \kappa(\lambda)$ is typically small in practice, leading to efficient vertex processing in real-world ML graphs.

Traversal Order Selection Strategy. Here, we present our strategies to select the traversal order based on the ML graph's properties. The main difference between DP-Enum and LP-Enum lies in the pruning order of invalid vertices. The DP-Enum strategy fixes the degree dimension k and incrementally increases the layer dimension λ , removing vertices that no longer satisfy the (k, λ) -constraint. Once no vertices remain for a given k , it proceeds to the next k value

in the grid. In contrast, LP-Enum fixes λ and increases k , pruning based on degree within each layer. After removing all vertices, it proceeds to the next λ value in the grid. This difference in pruning order can significantly impact algorithm performance, as the best order depends on the characteristics of the ML graph. As an example, consider the Homo dataset [23, 24] with 7 layers. Figure 4 shows the percentage of valid vertices remaining when pruning with $k = 1$ while increasing λ (left, using LP-Enum), and with $\lambda = 1$ while increasing k (right, using DP-Enum). The results reveal that LP-Enum can eliminate almost all vertices by the time $\lambda = 4$ and $k = 1$, whereas DP-Enum only removes approximately 50% of vertices when $k = 4$ and $\lambda = 1$, with $\lambda_{\max}$ equals to 5 and $k_{\max}$ equals to 39. This indicates that as λ increases, LP-Enum rapidly shrinks the subgraph corresponding to the $(1, \lambda)$ -FirmCore, leading to a smaller computational space for computing higher k values under the same λ , while DP-Enum maintains a much larger computational space as k grows. This effect is further reflected in the edge reduction. For the Homo dataset, the total number of edges removed from the computational space under LP-Enum is $\sum_{\lambda=1}^{\lambda_{\max}} |E_{1, \lambda-1}| = 467040$ whereas under DP-Enum it is $\sum_{k=1}^{k_{\max}} |E_{k-1, 1}| = 3167120$. Thus, LP-Enum saves approximately 6.78 $\times$ workload than DP-Enum on this dataset. Therefore, we select LP-Enum when FirmCore subgraphs shrink faster along the λ dimension than along the k dimension, which often correlates with $\lambda_{\max} < k_{\max}$. Conversely, we select DP-Enum when the subgraphs along the k dimension exhibit faster reduction, which is commonly associated with $k_{\max} < \lambda_{\max}$. In practice, a sparse ML graph with many layers benefits from DP-Enum while a dense ML graph with fewer layers benefits from LP-Enum.

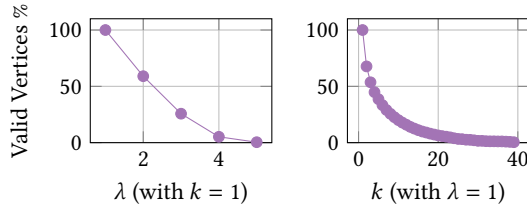


Figure 4: Percentage of valid vertices for Homo dataset.

Comparing SerGrid and CoreIndex [29]. In summary, our SerGrid introduces key innovations over CoreIndex in several aspects. First, it fully leverages the two-dimensional hierarchical of (k, λ) -FirmCore to reduce computation. Second, it stores the valid vertex set for each FirmCore explicitly, enabling efficient retrieval without additional processing. Third, it supports different traversal orders based on the graph's property to improve the overall efficiency. Finally, its bucket-free design avoids costly synchronization and atomic operations across buckets, forming the basis of our parallel CPU (Section 5.1-5.2) and GPU (Section 5.3) algorithms..

5 PARALLEL FIRM CORE DECOMPOSITION

Building on FC-Grid, our parallel algorithms fully exploit thread-level parallelism while minimizing the resource underutilization of prior methods [29]. To address the idle threads issue, we design a *vertex-level parallel strategy* that enables parallel execution across

Algorithm 2: ParGrid($k, \lambda, PValid, PCnt, deg$)

Data: $k, \lambda, PValid, PCnt, deg$; **Result:** FC-Grid $_{k, \lambda}$

```

1 if  $PCnt = G.vertex()$  then return
2  $pos_s \leftarrow 0, pos_e \leftarrow 0, buf \leftarrow \emptyset$   $\triangleright$  Information in each thread
3  $Count[\cdot] \leftarrow 0_{|V|}$   $\triangleright$  Initialize Count with all elements be 0
4 for  $v = 0$  to  $|V|$  - In parallel do
5   if  $PValid[v] = 1$  then
6      $cnt \leftarrow 0$ 
7     for  $l = 0$  to  $G.layer()$  do  $cnt \leftarrow cnt + deg_l(u) \geq k$ 
8      $Count[v] \leftarrow cnt$ 
9     if  $cnt < \lambda$  then  $buf[pos_e++] \leftarrow v, PValid[v] \leftarrow 0$ 
10 Barrier
11 while  $pos_s < pos_e$  - All threads work in parallel do
12    $v \leftarrow buf[pos_s++]$   $\triangleright$  Each thread gets a vertex
13   for  $l = 0$  to  $G.layer()$  do
14     for  $u \in N_l(v)$  do
15        $d \leftarrow atomicSub(deg_l(u), 1)$ 
16       if  $d = k - 1$  then
17          $c \leftarrow atomicSub(Count(u), 1)$ 
18         if  $c = \lambda - 1$  and  $atomicCheck(PValid(u))$  then
19            $buf[pos_e++] \leftarrow u$ 
20 Barrier
21  $atomicAdd(PCnt, pos_e)$ 
22 if  $threadID = 0$  then FC-Grid $_{k, \lambda} \leftarrow PValid$ 

```

vertices (Section 5.1). To improve thread utilization, we propose grouping a small set of threads for each FirmCore computation, allowing threads within a group to compute a single FirmCore in parallel, while multiple groups process multiple FirmCores concurrently. This strategy enables both *inter- and intra-FirmCore parallelism* (Section 5.2). Moreover, to enable additional intra-vertex parallelism, we propose a GPU-based solution (Section 5.3).

5.1 Vertex-level parallelism on FC-Grid

As previously observed, the parallelization solely across λ values in prior work [29] results in idle threads, since the number of layers is typically smaller than the number of available threads. This limitation is evident from Figure 1, which reports the speedup of ParCoreIndex with increasing thread counts. Beyond 10 threads, corresponding to the number of layers in the graph, additional threads yield no further performance gain.

CHALLENGE 1 (IDLE THREAD). *Can we eschew idle threads and thus harness the full CPU power?*

To address this issue, we present a *vertex-level* parallel strategy that builds upon the bucket-free FC-Grid structure. Yet, parallelizing the FC-Grid construction is non-trivial due to its sequential nature and the inter-dependencies in degree updates across layers. To tackle this problem, we adopt a *frontier-based* strategy [35], which stores the peeled vertices in each iteration and decouples the peeling (vertex removal) process from degree updates. In a nutshell, for each degree and layer constraint, we peel in parallel all vertices that do not belong to the (k, λ) -FirmCore and collect them into a frontier. Unlike a global frontier [5], which suffers from

thread contention and synchronization overhead, our design utilizes *thread-local frontiers*. By ensuring each vertex is processed by a single thread to update neighbor degrees, we eliminate frontier-level contention and maximize resource utilization.

Algorithm 2 presents the pseudocode for vertex-level parallel FirmCore computation under degree constraint k and layer constraint λ (ParGrid). $PValid$ is a binary array that records each vertex's status, with $PValid[v] = 1$ indicating that vertex v satisfies the (k, λ) -constraint. The variable $PCnt$ tracks the number of vertices removed from the ML graph for each FirmCore as those vertices violate the (k, λ) -constraint. If all the vertices are removed from the ML graph, it indicates that the decomposition for the corresponding k - or λ -dimension is complete (Line 1). Otherwise, each thread initializes a local buffer and checks whether each vertex satisfies the (k, λ) -constraint in parallel; if not, the thread adds the vertex to its own buffer and sets the vertex's valid status to false. In the end, each thread's buffer stores the vertex that does not satisfy the constraint (Line 4-9). Next, all the threads work in parallel with vertex-level parallelism as illustrated in Figure 5; each thread gets a vertex v from its own buffer and removes its connection with its neighbor u across layers. Each degree update must be atomic to prevent a situation in which multiple threads concurrently modify a vertex's degree (Line 15). After an update, each thread inspects whether the degree still satisfies the threshold k for that layer. If not, the thread atomically decrements the variable $Count$, which records the number of layers in which the vertex has degree at least k (Line 17). When $Count$ becomes $\lambda-1$, the vertex no longer satisfies the (k, λ) constraint. In this case, we atomically set its validity to 0 and insert the vertex into the thread-local buffer (Line 18-19). To prevent an already peeled vertex from being added to the thread-local buffers again, we employ an atomic check (`atomicCheck`) on its validity status. Specifically, if a vertex u has $PValid[u]=1$, `atomicCheck` sets $PValid[u]$ to 0 and returns true; otherwise, if u 's validity is already 0, it returns false. This process repeats until all threads' buffers are empty. Finally, we update $PCnt$ with the number of newly removed vertices from each buffer (Line 21) and store the valid status of each vertex $PValid$ in FC-Grid (Line 22). As shown in Figure 3, in practice, each FirmCore receives the $PValid$ and the $PCnt$ value from its parent FirmCore as intermediate results to avoid redundant computation and improve efficiency.

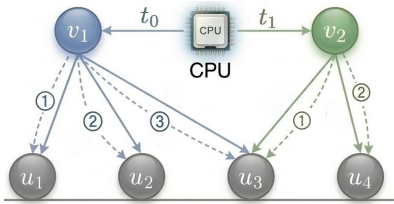


Figure 5: Vertex-level parallelism within a single layer. CPU threads t_0 and t_1 concurrently process vertices v_1 and v_2 , respectively. Each thread sequentially scans its assigned neighborhood: t_0 processes u_1, u_2, u_3 , while t_1 processes u_3, u_4 .

Time complexity. We analyze our parallel algorithms under the work-span model [33]. The work of an algorithm is the total number of operations, while span is the longest dependency path.

THEOREM 2. *Our SerGrid equipped with the ParGrid in Algorithm 2 finds all non-empty FirmCore in $O(|L| \cdot |V| \cdot \sum_{\lambda=1}^{\lambda_{\max}} \kappa(\lambda) + |L| \cdot |E|)$ work and $O(\sum_{\lambda=1}^{\lambda_{\max}} \kappa(\lambda))$ span.*

PROOF. SerGrid with ParGrid computes the FirmCore for each $\lambda \in [1, \lambda_{\max} < |L|]$. For each λ , the algorithm performs $\kappa(\lambda)$ peeling iterations. Recall that $\kappa(\lambda)$ is the largest value such that $(\kappa(\lambda), \lambda)$ -FirmCore is non-empty. In each iteration, ParGrid inspects all layers for valid vertices and removes edges associated with invalid vertices. As each edge is removed only once, it takes $O(\kappa(\lambda) \cdot |L| \cdot |V| + |E|)$ to compute all FirmCores that share the same λ value. Since there are $\lambda_{\max}$ lists, the work is $O(|L| \cdot |V| \cdot \sum_{\lambda=1}^{\lambda_{\max}} \kappa(\lambda) + |L| \cdot |E|)$. For each λ , ParGrid performs up to $\kappa(\lambda)$ iterations. Since ParGrid inspects $\lambda_{\max}$ layers, the span is $O(\sum_{\lambda=1}^{\lambda_{\max}} \kappa(\lambda))$. $\square$

Reducing memory redundancy. Our ParGrid improves the efficiency of FirmCore decomposition by addressing the challenge of wasting computational resources. However, a key challenge lies in the memory consumption associated with storing the $PValid$ array in each FirmCore. This array, with a length equal to the number of vertices, becomes especially memory-consuming for large values of k and λ , wherein only a small subset of vertices satisfy the constraints. To address this issue, instead of storing the entire $PValid$ array, we store only the valid vertices for each FirmCore. For example, Table 1(a) illustrates the valid vertex sets for each (k, λ) combination based on Figure 2 (Top). However, this approach still introduces some memory redundancy. Specifically, certain vertex sets are stored multiple times across different FirmCores. For instance, both the $(1, 1)$ -FirmCore and the $(2, 1)$ -FirmCore contain the same set of vertices: $\{0, 1, 2, 3, 4, 5\}$. To further reduce memory usage, we adopt a differential storage strategy [43–45]. Rather than storing the complete valid vertex set for each FirmCore, we store the difference between a FirmCore and its child. For example, in Table 1(b) the $(1, 1)$ -FirmCore stores only the difference between itself and its child $(2, 1)$ -FirmCore. Since these two FirmCores share the same vertex set, the difference is empty, and thus $(1, 1)$ -FirmCore stores an empty set. Similarly, the $(2, 1)$ -FirmCore stores its difference with $(3, 1)$ -FirmCore, which is $\{2, 5\}$. Finally, since $(3, 1)$ -FirmCore has no child, it retains its full valid vertex set: $\{0, 1, 3, 4\}$. To retrieve the valid vertex set given k and λ , we can traverse our FC-Grid to aggregate the vertices. For example, to retrieve $(1, 1)$ -FirmCore, we traverse along the $\lambda=1$ dimension and aggregate all the vertex sets corresponding to $k \geq 1$. Thus, $(1, 1)$ -FirmCore = $\{\} \cup \{2, 5\} \cup \{0, 1, 3, 4\} = \{0, 1, 2, 3, 4, 5\}$.

THEOREM 3. *For an ML graph $G = (V, E, L)$, our ParGrid with the differential storage strategy stores all FirmCores in $O(\sum_{\lambda=1}^{\lambda_{\max}} |V_{1,\lambda}|)$ space, where $V_{1,\lambda}$ is the vertex set of the $(1, \lambda)$ -FirmCore.*

PROOF. ParGrid with the differential storage strategy computes the FirmCores for each $\lambda \in [1, \lambda_{\max}]$. For a fixed λ , since $(k+1, \lambda)$ -FirmCore is a subset of (k, λ) -FirmCore, ParGrid stores, for each k , only the set difference $V_{k,\lambda} \setminus V_{k+1,\lambda}$. These difference sets are pairwise disjoint for a fixed λ , and their union over all k is $V_{1,\lambda}$. As such, the space to store all FirmCores is $O(\sum_{\lambda=1}^{\lambda_{\max}} |V_{1,\lambda}|)$. $\square$

ParGrid is more parsimonious than CoreIndex and ParCoreIndex, which requires $O(\lambda_{\max} \cdot |V|)$ space to store the core index of each

	$\lambda = 1$	$\lambda = 2$	$\lambda = 3$
$k = 1$	{0, 1, 2, 3, 4, 5}	{0, 1, 2, 3, 4, 5}	{0, 1, 2, 3, 4, 5}
$k = 2$	{0, 1, 2, 3, 4, 5}	{0, 1, 2, 3, 4}	{}
$k = 3$	{0, 1, 3, 4}	{0, 1, 3, 4}	{}

(a) Storing the valid vertex set

	$\lambda = 1$	$\lambda = 2$	$\lambda = 3$
$k = 1$	{}	{5}	{0, 1, 2, 3, 4, 5}
$k = 2$	{2, 5}	{2}	{}
$k = 3$	{0, 1, 3, 4}	{0, 1, 3, 4}	{}

(b) Storing the difference

Table 1: Vertex set storing strategies for graph in Fig. 2.

vertex [29]. In contrast, we store all FirmCores in $O(\sum_{\lambda=1}^{\lambda_{\max}} |V_{1,\lambda}|)$ space. Since the size of the $(1, \lambda)$ -FirmCore typically decreases as λ increases and $|V_{1,\lambda}| \leq |V|$, the space consumption $\sum_{\lambda=1}^{\lambda_{\max}} |V_{1,\lambda}|$ is significantly smaller than $\lambda_{\max} \cdot |V|$ in practice.

5.2 Mitigating thread underutilization

Besides idle threads, we also observe previous methods suffering from thread underutilization due to unbalanced workload across threads [29]. For instance, in the example in Figure 1, increasing the number of threads from 6 to 8 yields imperceptible speedup on the HW1 dataset. This is because the graph has 10 layers, so configurations with 6 to 8 threads require two processing rounds. In these cases, some threads process two lists while others handle one, leaving part of the threads underutilized and consequently preventing any performance improvement.

CHALLENGE 2 (THREAD UNDERUTILIZATION). *Can we balance the workload to prevent thread underutilization?*

While ParGrid addresses the idle thread issue by parallelizing the computation within a single FirmCore using multiple threads, an effective strategy for FirmCores with a large size, we still experience low thread utilization for small FirmCores. As k or λ increases, the size of the FirmCore decreases, leading to reduced per-thread workload, rendering a large number of threads (e.g., 32 or 40) inefficient due to limited per-thread workload.

Thread Grouping Strategy. To further increase thread utilization in ParGrid, we introduce a thread grouping strategy that aggregates threads to collaboratively process each FirmCore, while different groups operate in parallel on distinct FirmCores. This design enables both intra- (i.e., vertex-level) and inter-FirmCore (i.e., FirmCore-level) parallelism, a version we denote as ParGrid+. This moderate level of intra-FirmCore parallelism maintains high thread utilization even when the FirmCores’ size varies. On a machine with T threads, ParGrid+ partitions them into groups of size t , enabling $\frac{T}{t}$ FirmCore to compute in parallel. Here, t is a tunable parameter that controls how many threads collaborate on computing each FirmCore using ParGrid as a building block. When $t = T$, ParGrid+ behaves like ParGrid, where all threads collaborate on a single FirmCore. By tuning t , ParGrid+ mitigates the thread underutilization issues when computing FirmCore with small FirmCores.

Moreover, assigning fewer threads to each FirmCore does not reduce the overall parallelism of ParGrid; instead, it allows more FirmCores to compute in parallel with higher thread utilization.

5.3 Parallelizing vertex neighbor processing on the GPU

Our algorithms allow multiple vertices to work in parallel, but each vertex still has to update its neighbors’ degrees sequentially due to the limited number of threads available.

CHALLENGE 3 (LIMITED INTRA-VERTEX PARALLELISM). *Can we improve vertex-level parallelism while reducing sequential per-vertex computation?*

Yet, the CPU algorithm already operates near its hardware limits, as the number of physical threads equals the number of cores. Given this constraint, we turn our attention to GPUs, which offer massive numbers of lightweight threads and enable fine-grained parallelism. However, designing a GPU-based FirmCore decomposition algorithm is non-trivial due to GPU-specific challenges such as memory coalescing, limited per-thread memory, and thread divergence.

ML graph representation on GPU. GPUs pose unique challenges for representing ML graphs due to their hierarchical memory architecture. Thus, we load the entire ML graph into global memory using a degree array of size $|V| \times |L|$, where each contiguous space of size $|L|$ stores the degree information of a vertex across all layers. Since the degrees of vertices are stored in a continuous manner, this layout facilitates thread-coalesced access, improving memory access efficiency during parallel computation. Figure 6 shows the degree representation of Figure 2 on a GPU. Then, we represent each layer of the ML graph using the CSR format [28], which stores the offset array and the corresponding adjacency list efficiently.



Figure 6: ML graph degree representation on GPU.

Frontier design. On the CPU, each thread maintains its own buffer for storing invalid vertices. Yet, this approach is not feasible on GPUs because each thread has scarce local memory, and independent buffer management becomes highly inefficient under SIMT execution, where all threads within a warp must execute the same instruction. However, a shared global frontier incurs significant thread contention and synchronization overhead due to concurrent vertex insertions. To overcome this limitation, we organize threads into blocks, each block maintaining a frontier (*buf*) shared across all threads within that block, allowing parallelism among vertices in different thread blocks. Each block also maintains a corresponding counter, *pos_e*, which tracks the number of invalid vertices stored in its buffer. Given the degree constraint k and the layer constraint λ , all threads operate in parallel to check whether their corresponding vertices satisfy the constraints following Algorithm 2; if not, the thread adds it to its block’s buffer, thereby effectively reducing thread contention. Specifically, we use the first thread of each block to initialize the shared variable *sh_pos_e*, which tracks the number of

invalid vertices that do not satisfy the (k, λ) -constraint in its buffer. We allocate sh_pos_e in shared memory due to its lower latency compared to global memory. In the end, the first thread of each block writes the value of sh_pos_e back to global memory pos_e .

Hierarchical parallelism. Directly applying vertex-level parallelism on GPUs is inefficient. This is because real-world graphs often exhibit skewed degree distributions, leading to significant warp divergence among threads. To address the limitation and further increase parallelism, we propose the hierarchical parallelism strategy that enables fine-grained cooperation among threads for each vertex, as illustrated in Figure 7. Specifically, vertices in the frontier are processed in parallel with a group of threads works in parallel to update the degree of the neighbors of each vertex, enabling both inter-vertex and intra-vertex parallelism. As all threads within a warp follow the same update operations, our strategy alleviates intra-warp thread loop divergence and ensures efficient SIMT execution.

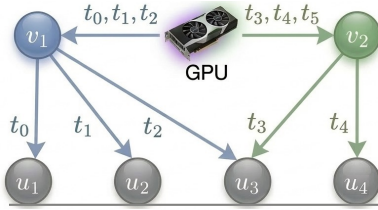


Figure 7: Hierarchical parallelism within a single layer; the GPU launches a group of threads to process each peeled vertex v_1 and v_2 . For v_1 , its neighbors u_1 , u_2 , and u_3 are processed in parallel by threads t_0 , t_1 , and t_2 . For v_2 , its neighbors u_3 and u_4 are processed in parallel by threads t_3 and t_4 . This enables parallel processing at both vertex and neighbor levels.

The work and span of our GPU version are analogous to those of the CPU versions.

Discussion. Building on FC-Grid, we design three parallel algorithms, ParGrid, ParGrid+, and ParGrid++, providing insights into parallel design for improved computational resource utilization. While offline analysis of ML graphs may tolerate longer runtime when FirmCore decomposition is performed only once, our algorithms improve the efficiency and scalability of FirmCore decomposition, particularly for billion-scale ML graphs. This makes FirmCore analysis more practical and responsive in fast-turnaround scenarios, such as interactive exploration of FirmCore structures or repeated decomposition as the ML graph evolves.

6 EXPERIMENTS

We run our CPU algorithms on a 2-socket Intel® Xeon® Silver 4410T processor with 20 cores and 252GB RAM, using OpenMP and GCC 11.4.0, and the GPU algorithms on an NVIDIA GeForce RTX 4090 with 24GB memory using CUDA and nvcc 12.2.14.

Datasets. We perform experiments on 10 real-world datasets [10, 16–18, 23, 37, 40, 41, 44, 51, 53] from various domains. We additionally generate two sparse datasets, EM and LJ, from Email-EuAll and LiveJournal [41]. We consider the original graph as layer 0 and construct additional layers by sampling edges incrementally: for EM, we

add edges in 20% increments (layers 1–4), and for LJ 30% increments (layers 1–2). We further generate two dense billion-scale datasets, *Hollywood-1* and *Hollywood-2*, from the Hollywood-2011 [7, 8] actor-actor graph, using the original graph as layer 0. *Hollywood-1* samples edges in 10% increments (layers 1–9), and *Hollywood-2* in 5% increments (layers 1–18). Table 2 presents their characteristics.

Table 2: Datasets; number of nodes $|V|$, edges $|E|$, layers $|L|$; smallest $\min_l |E_l|$, largest layer $\max_l |E_l|$; domain.

Graph	$ V $	$ E $	$ L $	$\min_l E_l $	$\max_l E_l $	Domain
(H) Homo	18k	153k	7	256	83k	genetic
(S) Sacchcere	6.5k	247k	7	1.3k	91k	genetic
(F) FAO	214	319k	364	1	6.5k	financial
(I) Internet	62k	506k	9	20k	147k	P2P
(W) Wiki	1.0M	2.9M	10	347	988k	social
(OI) ObamaInIsrael	2.2M	3.8M	3	557k	1.8M	social
(A) Amazon	410k	8.1M	4	899k	2.4M	purchase
(DL) DBLP-Large	1.8M	10M	22	107	2.6M	author
(FG) FlickrGrowth	2.3M	23M	6	121k	13M	social
(SO) StackOverflow	2.6M	47.9M	24	172k	2.9M	social
(EM) Email-EuAll	25.6k	728k	5	73k	364k	communic.
(LJ) LiveJournal	4.8M	68.5M	3	12.8M	42.8M	social
(HW1) Hollywood-1	2.0M	0.62B	10	22.9M	114.49M	actors
(HW2) Hollywood-2	2.0M	1.2B	19	22.9M	114.49M	actors

Algorithms. We test the following algorithms:

- (1) ParMLCD [44]: The state-of-the-art parallel k -core decomposition algorithm on ML graphs. We use all optimizations with 40 threads and hyperparameter $\alpha = 0.1$, $l = 3$.
- (2) CoreIndex [29]: Reimplementation of the core index-based state-of-the-art FirmCore decomposition algorithm in C++ for fair comparison, achieving 1.1× to 4.83× speedup over the original Python version [29].
- (3) ParCoreIndex [29]: Reimplementation of the parallel CoreIndex in C++ with OpenMP achieving at least 2× speedup over the original Python version [29].
- (4) SerGrid: Our serial grid-based algorithm that selects between DP-Enum and LP-Enum for efficient FirmCore decomposition, based on the structural properties of the ML graph (Section 4).
- (5) ParGrid: Our FC-Grid with vertex-level parallel strategy for FirmCore decomposition on multi-core CPUs (Section 5.1).
- (6) ParGrid+: Our FC-Grid with intra- and inter-FirmCore decomposition on multi-core CPUs (Section 5.2).
- (7) ParGrid++: Our GPU algorithm for FirmCore decomposition (Section 5.3).

6.1 Runtime

We first study the runtime of all algorithms (y-axis in log scale).

SerGrid, CoreIndex and ParMLCD. We initially compare the runtime of the CoreIndex with our serial algorithm SerGrid (Figure 8). Across all datasets, SerGrid achieves speedups of 1.4× to 19.64×, with an average of 9.1× over CoreIndex. This demonstrates the efficiency of our two-dimensional grid, which computes each FirmCore from its parent’s valid vertex set, reducing redundant computation. We then compare SerGrid with ParMLCD, the state-of-the-art algorithm for k -core decomposition on ML graphs. As ParMLCD does not directly support FirmCore decomposition, it first computes

the k -cores, after which we add each non-empty k -core to the corresponding (k, λ) -FirmCore set. On F, SO, FG, HW1, and HW2, ParMLCD does not terminate within 5 hours. On the remaining datasets, SerGrid achieves speedups of 1.6 $\times$ to 248 $\times$ over ParMLCD.

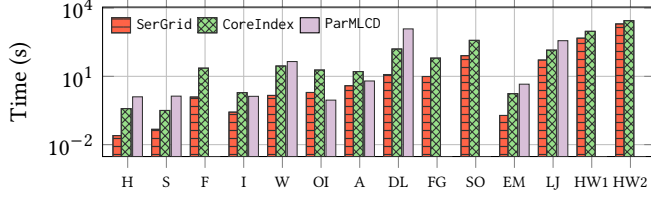


Figure 8: Runtime of SerGrid, CoreIndex, and ParMLCD.

DP-Enum and LP-Enum. We then compare our two traversal strategies DP-Enum and LP-Enum in Figure 9. SerGrid chooses between DP-Enum and LP-Enum according to the ML graph’s property. LP-Enum is faster than DP-Enum on most datasets, outperforming DP-Enum on the FG, HW1 and HW2 datasets with 20.1 $\times$, 13.5 $\times$ and 26.5 $\times$ speedup, respectively. Only on F, LP-Enum is approximately 5.7 $\times$ slower than DP-Enum.

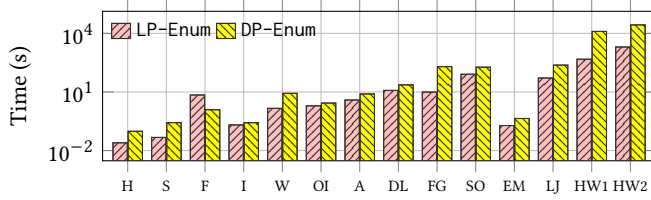


Figure 9: Runtime comparison of LP-Enum and DP-Enum.

To further interpret these results, Table 3 reports the values of $k_{\max}$, and $\lambda_{\max}$ across these datasets. In F dataset, $\lambda_{\max}$ is about 6 $\times$ larger than $k_{\max}$. Therefore, using the LP-Enum strategy for FirmCore decomposition would require 336 constructing rounds. In contrast, the DP-Enum strategy requires only 51 constructing rounds, with each round having smaller computational space, thereby offering a substantial performance advantage. Figure 10 further corroborates this argument by comparing the total number of edges removed using LP-Enum and DP-Enum. For most datasets, LP-Enum incurs a smaller workload, whereas the F dataset benefits more from DP-Enum. This reasoning also applies to other datasets. We thus recommend using the DP-Enum algorithm in a sparse ML graph with a large number of layers, such as the F dataset. Conversely, dense ML graphs with fewer layers tend to benefit more from the LP-Enum algorithm, such as the FG, SO, HW1, and HW2 datasets. We also test layer ordering by graph density and edge count; results show minimal impact on efficiency.

Table 3: $k_{\max}$ and $\lambda_{\max}$ from all datasets.

	H	S	F	I	W	OI	A	DL	FG	SO	EM	LJ	HW1	HW2
$k_{\max}$	39	54	51	10	72	38	10	286	506	79	4	30	1297	1297
$\lambda_{\max}$	5	7	336	9	8	3	4	15	6	22	3	372	10	19

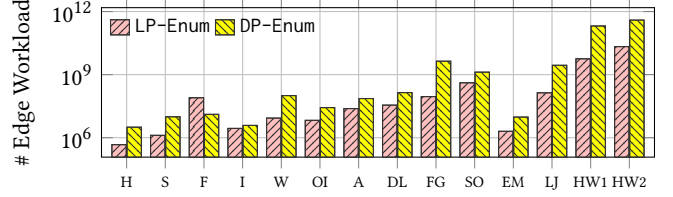


Figure 10: Removed edges with LP-Enum and DP-Enum.

Parallel Algorithms. We evaluate our parallel algorithms and show that our parallel strategy enhances efficiency (Figure 11). We first compare our parallel algorithm with the state-of-the-art parallel algorithm ParCoreIndex. Our algorithms, ParGrid and ParGrid+ outperform ParCoreIndex by up to 30.3 $\times$, and 33.7 $\times$, with respective average speedups of 7.5 $\times$, and 10.8 $\times$, under the same thread counts. This highlights the effectiveness of our algorithm design through better utilization of the computational resources. We then juxtapose our parallel algorithms with our serial algorithm SerGrid. ParGrid achieves speedups ranging from 1.3 $\times$ to 18.8 $\times$ compared to SerGrid. By improving the thread utilization, our ParGrid+ further enhances efficiency, particularly on large datasets, achieving more than 2 $\times$ speedup over both ParGrid on billion-scale HW1 and HW2 using the same thread counts, and outperforms SerGrid by more than 35 $\times$. Similarly, ParGrid+ significantly benefits datasets with a large number of layers; on the F dataset, it achieves a speedup of more than 5 $\times$ compared to ParGrid.

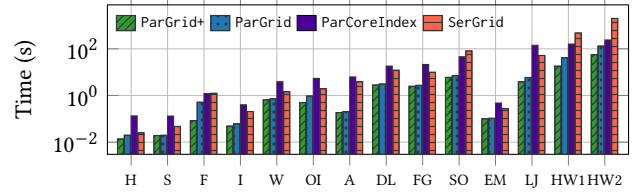


Figure 11: Runtime comparison of CPU parallel algorithms.

Moreover, compared to the state-of-the-art serial CoreIndex, ParGrid achieves speedups ranging from 19.4 $\times$ to 78.8 $\times$. Our hybrid strategy, ParGrid+, further enhances performance, achieving speedups between 28.0 $\times$ and 275.7 $\times$, with an average of 63.1 $\times$.

Effectiveness of Our GPU-Based Approach. Figure 12 compares ParGrid, ParGrid(GPU), ParGrid++, and ParCoreIndex algorithms. ParGrid(GPU) is a naïve GPU adaptation of ParGrid, which adopts vertex-level parallelism by assigning a GPU thread to each vertex. Despite the larger degree of parallelism available on GPUs, ParGrid(GPU) achieves performance comparable to, or even worse than ParGrid on datasets such as HW1 and HW2, primarily due to severe thread divergence. In contrast, our GPU-optimized ParGrid++ effectively mitigates this issue and achieves a speedup of up to 6.32 $\times$ over ParGrid(GPU). Furthermore, ParGrid++ runs up to 26.2 $\times$ faster than ParGrid with on average of 8.2 $\times$. This demonstrates that our GPU design with a hierarchical parallel strategy effectively increases the parallelism and substantially improves efficiency. Moreover, ParGrid++ runs in 0.24 seconds on average,

enabling more responsive exploration [48] of the FirmCore structure. In contrast, the state-of-the-art parallel baseline run in 45 seconds on the million-scale dataset SO. Furthermore, ParGrid++ outperforms the parallel ParCoreIndex baseline by up to 796 $\times$, with an average 100.3 $\times$ speedup.

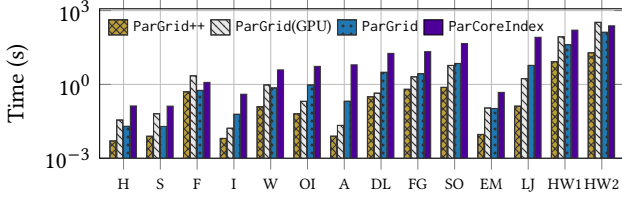


Figure 12: Effectiveness of GPU algorithm design.

Scalability in #Layers. We study the scalability of FirmCore decomposition algorithms with the number of layers, reporting only results for the 4 largest datasets in Fig. 13 due to space limitations. Our ParGrid++ achieves the highest efficiency across all layer settings, delivering average speedups of 161.6 $\times$ over CoreIndex and 44.1 $\times$ speedup over ParCoreIndex. Our serial algorithm SerGrid consistently outperforms the serial baseline CoreIndex with an average 4.1 $\times$ speedup. Our CPU parallel algorithms ParGrid and ParGrid+ achieve 6.8 $\times$ and 11.4 $\times$ speedup on average over the parallel baseline ParCoreIndex under the same thread counts. By improving thread utilization, ParGrid+ further improves efficiency on large-scale datasets such as SO and billion-scale HW1 and HW2, under the same thread count, achieving 2.2 $\times$ speedups over ParGrid. Thus, our algorithms scale gracefully in the number of layers.

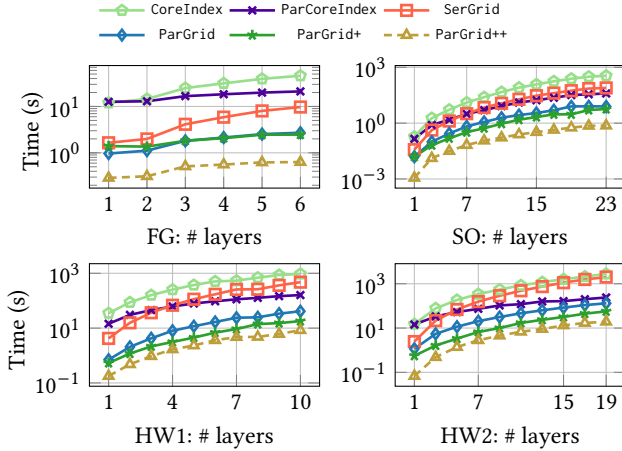


Figure 13: Runtime vs. number of layers (log-scale y-axis)

Scalability in number of threads. Figure 14 compares the scalability of our ParGrid and the state-of-the-art ParCoreIndex with increasing number of threads from 1 to 40 on the largest, million-scale SO and billion-scale HW1 and HW2. Our ParGrid scales gracefully in the number of threads, and consistently outperforms ParCoreIndex under the same thread count. In contrast, ParCoreIndex

stops improving when the number of threads exceeds the number of layers in the ML graph, since it can only parallelize over the layers. This result vindicates our design choices in ParGrid, demonstrating improved scalability and more effective utilization of computational resources for higher efficiency.

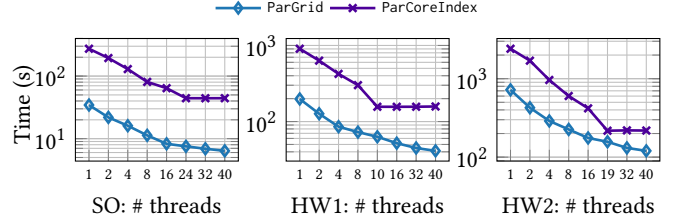


Figure 14: Runtime comparison of ParGrid and ParCoreIndex with increase # thread.

We then study the runtime of ParGrid+ and ParGrid with increasing the thread counts. For ParGrid+, we set $t = 8$ threads for each FirmCore computation and vary the number of FirmCore computations in parallel from 1 to 5. As shown in Figure 15, increasing the number of threads improves the performance of ParGrid, but the gain diminishes with more threads (e.g., 32 and 40) due to relatively low thread utilization. In contrast, ParGrid+ assigns groups of $t = 8$ threads to each FirmCore computation and enables multiple FirmCores to compute in parallel, thereby increasing thread utilization. Thus, under the same thread counts, ParGrid+ consistently outperforms ParGrid with around 2 $\times$ speedup.

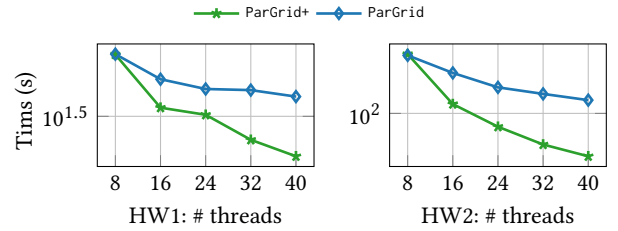


Figure 15: Runtime in number of threads: ParGrid+, ParGrid

Thread Utilization. We compares the thread utilization achieved by the different parallel algorithms. We define thread utilization as $U = \frac{\sum_r \sum_{i=1}^{P_r} T_{i,r}^{coer}}{\sum_r P_r \cdot T_r^{wall}}$, where r indexes a parallel region (i.e., a region where a fixed number of threads run in parallel), P_r denotes the number of threads used in region r , $T_{i,r}^{coer}$ is the effective computation time of thread i in region r , and T_r^{wall} is the wall-clock execution time of region r . Thus, $U \in [0, 100\%]$, with 100% indicating full utilization. We use $P_r = 20$ to exclude hyper-threading effects. As shown in Figure 16, ParCoreIndex generally shows low utilization because many threads remain idle. An exception is dataset F, which consists of 364 layers: in this case, ParCoreIndex achieves high utilization by assigning each thread to process a different layer independently. By contrast, ParGrid improves utilization by exploiting vertex-level parallelism, which leads to a more balanced distribution of work across threads. Finally, ParGrid+ further increases

thread utilization beyond ParGrid by combining both intra- and inter-FirmCore parallelism, enabling more effective concurrency throughout the computation.

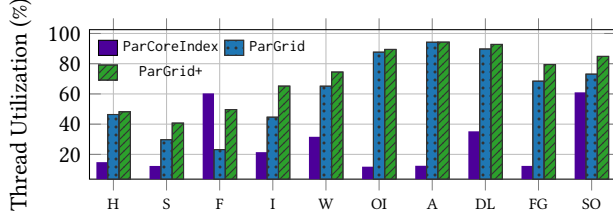


Figure 16: Thread utilization of the parallel algorithms.

6.2 Memory

SerGrid and CoreIndex. We first compare our serial SerGrid with CoreIndex. As shown in Figure 17, across all datasets, SerGrid consumes $2.7\times$ less memory on average. This is mainly because CoreIndex maintains, for each λ , a bucket structure sized by the graph’s maximum degree and uses set-based structures for vertex lookup and deletion, incurring memory overhead. In contrast, SerGrid avoids bucket maintenance, achieves a smaller memory footprint, and delivers an average $9.1\times$ speedup. Although FC-Grid stores the valid vertex set for each (k, λ) pair, increasing memory use, it enables efficient FirmCore retrieval.

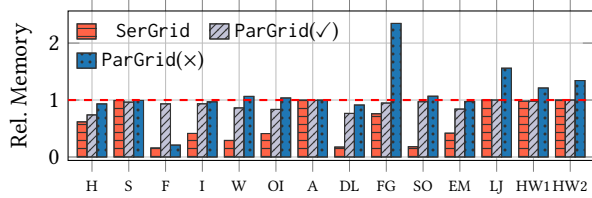


Figure 17: Relative memory consumption of SerGrid, ParGrid(✓) and ParGrid(×) normalized to the baseline CoreIndex (red dotted line). Lower is better.

Parallel algorithms. We analyze the memory consumption of ParGrid, with, ParGrid(✓), and without, ParGrid(×), the memory-saving strategy. We omit ParGrid+, which differs only in parallelization and uses the same memory as ParGrid. As shown in Figure 17, ParGrid(×) consumes more memory than SerGrid because it maintains additional degree information, but remains comparable to CoreIndex on most datasets. In particular, it maintains a vertex-sized visit array to track validity for every (k, λ) -FirmCore. On FG and LJ, this array accounts for nearly half of the total memory, causing higher usage than CoreIndex. To reduce this overhead, ParGrid(✓) stores only valid vertices and applies the hierarchical memory-saving strategy in Section 5.1. Table 4 shows that this strategy reduces memory consumption by 30.5% on average.

ParGrid++ peak memory consumption. We report the memory consumption of the input graph and the peak GPU memory usage of

Table 4: Memory consumption (in MB) with and without memory-saving optimization.

	H	S	F	I	W	OI	A	DL	FG	SO	EM	LJ	HW1	HW2
ParGrid(×)	13	12	110	51	934	682	197	4379	2612	9094	95	4231	18795	38163
ParGrid(✓)	11	12	109	46	702	507	197	3497	1012	7900	82	2714	15086	28211

ParGrid++ in Table 5. We store the input graph of each layer in the Compressed Sparse Row (CSR) format. We observe that ParGrid++ uses most of its GPU memory to store the input graph. Due to the limited capacity of GPU global memory, we transfer the result of each (k, λ) -FirmCore from the device to the CPU, thereby incurring no additional GPU global memory usage.

Table 5: Peak memory consumption (in MB) of ParGrid++.

	H	S	F	I	W	OI	A	DL	FG	SO	EM	LJ	HW1	HW2
G	2.25	2.35	2.77	14	112	90	84	398	292	780	28	738	4882	9438
ParGrid++	2.98	2.61	3.09	18	170	148	96	576	376	1050	30	778	4982	9606

7 FIRMCore APPLICATION STUDIES

7.1 FirmCore Retrieval

A practical use case of the FirmCore decomposition is to retrieve a particular FirmCore given k and λ ; e.g., in community detection, to retrieve a densely connected subgroup, or in biology, to retrieve a group of functionally related proteins. To evaluate the efficiency of our algorithms, we issue queries with different k and λ for all FirmCores on all datasets (Table 2). Figure 18 compares the total runtime to retrieve all FirmCores by traversing FC-Grid using LP-Enum with CoreIndex under both online and offline modes. The online strategy scans the entire vertex set to identify vertices that belong to the (k, λ) -FirmCore. Thus, its query time is linear in the number of vertices. The offline strategy first sorts the vertices by their core index for each layer, and then collects vertices that are part of the (k, λ) -FirmCore. Thus, the offline strategy avoids scanning the whole vertex set, improving efficiency, especially for large datasets. The sorting step incurs a preprocessing cost of $O(|L| \cdot |V| \cdot \log(|V|))$, but is only performed once. Our FC-Grid outperforms the offline query strategy by 4 orders of magnitude on average, because the query time of our FC-Grid is linear in the number of FirmCores. As shown in Table 6, the number of FirmCores in real-world datasets is much smaller than that of the vertices.

Table 6: Number of FirmCore

	H	S	F	I	W	OI	A	DL	FG	SO	EM	LJ	HW1	HW2
#FirmCore	58	101	3434	29	166	62	25	405	690	332	76	590	6664	12913

7.2 Temporal Social Network Analysis

We study how FirmCore can aid real-world temporal social network analysis, by improving the speed and scalability of FirmCore analysis in ML social networks. Specifically, we apply our efficient FirmCore decomposition algorithm to the Google+ dataset ¹ [27],

¹Google+ dataset download: <https://people.duke.edu/~zg70/gplus.html>

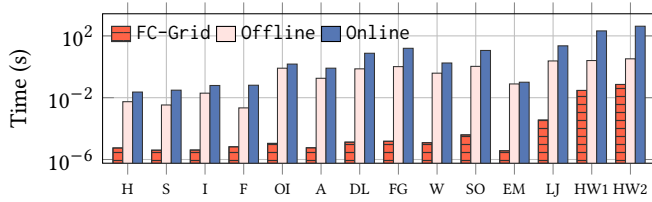


Figure 18: Runtime to retrieve all valid FirmCores.

which is an evolving social network with 28.9 million vertices, over 1.19 billion edges, and 4 layers, with each layer corresponding to a network snapshot at a specific time.

Insight. We first analyze the behavior of FirmCore under different λ values in Figure 19. Specifically, the maximum k decreases from 1572 when $\lambda = 1$, to 1559, 1035, and 204 for $\lambda = 2, 3$, and 4, respectively. This trend reflects the increasing strictness of the temporal constraint: as λ increases, only vertices that consistently maintain high connectivity across multiple timestamps are preserved. We further compute the k -core [1] on the aggregated temporal graph for comparison. The largest k is 1572 with 3223 vertices, which coincides with the result of FirmCore when $\lambda = 1$, as expected. However, when enforcing stronger temporal consistency (e.g., $\lambda = 4$), FirmCore identifies, with $k = 204$, a smaller subgraph with 2741 vertices. Despite being smaller, this subgraph captures more meaningful structures, as it consists of vertices that persist [30] across all timestamps rather than being dense only in a single snapshot. This result demonstrates that FirmCore enables the discovery of more stable and temporally consistent communities, which cannot be effectively identified by methods that ignore cross-layer constraints. Previous studies have shown that incorporating temporal information in graph analysis can benefit applications such as social network marketing [56]. Therefore, FirmCore has the potential to support such applications by identifying groups that are not only structurally cohesive but also persistent over time.

Efficiency. Importantly, this case study highlights the computational challenges of analyzing billion-scale ML graphs and underscores the need for efficient algorithm design. As shown in Figure 20, the state-of-the-art serial baseline CoreIndex requires more than 34 minutes, while the parallel baseline ParCoreIndex takes over 15 minutes. In contrast, ParGrid++ runs in approximately 5 seconds, achieving speedups of 368 $\times$ over CoreIndex and 156 $\times$ over ParCoreIndex. Overall, these results demonstrate that our approach enables fast and scalable temporal network analysis, making it practical for large-scale real-world datasets.

7.3 Multilayer Densest Subgraph Extraction

ML core decomposition also enables the extraction of dense subgraphs from ML graphs. Dense subgraphs have many applications, such as community detection [13] and link spam detection [39, 60], among others. FirmCore decomposition selects the FirmCore maximizing the density defined in [24], providing an approximate solution with a theoretical lower bound approximation ratio of $\frac{\psi_\beta}{2|L|^{\beta+1}}$ [29]. Here, we show that our parallel FirmCore algorithm accelerates densest subgraph extraction.

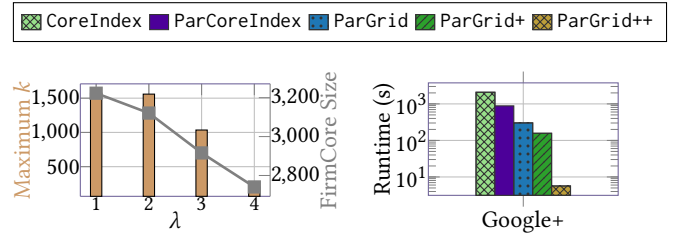


Figure 19: Maximum k of FirmCore for different λ . Figure 20: Runtime of FirmCore decomposition on Google+.

Figure 21 compares the runtime and density of our FC-Grid with ParMLCD for identifying the densest subgraph with $\beta = 2$. Here, β controls the trade-off between density and the number of layers sustaining it [23, 24], without affecting our algorithm’s performance. FC-Grid is based on the FirmCore model [29], while ParMLCD is built upon the ML k -Core model [24]. Overall, FirmCore achieves comparable or higher density than ParMLCD. Although ParMLCD identifies denser subgraphs on some datasets (e.g., OI and A), it suffers from severe efficiency limitations. In particular, it fails to find the densest subgraph within 5 hours on large datasets such as FG and SO and on ML graphs with many layers, such as F.

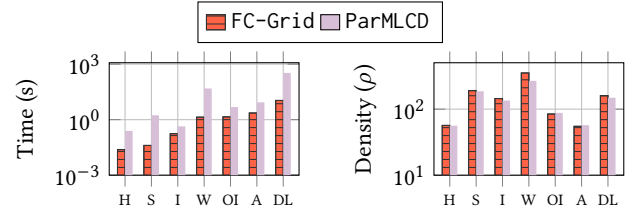


Figure 21: Runtime (left) and density (right) comparison for multilayer densest subgraph mining.

8 CONCLUSION

We propose a novel two-dimensional grid structure for efficient FirmCore decomposition. Our algorithm leverages the hierarchical structure of the FirmCore and two traversal strategies to reduce redundant computations that are easy to select based on simple properties of real-world ML graphs. Building on the grid structure, we develop parallel algorithms for multi-core CPUs and GPUs. Experiments on 14 datasets show that our serial and parallel algorithms outperform the state-of-the-art counterparts by 9 $\times$ and 100.3 $\times$ on average, respectively. Moreover, our method improves memory efficiency by 2.7 $\times$ on average. Our algorithms scale well with increasing layer and thread counts, making them suitable for large real-world ML graphs. Furthermore, FC-Grid accelerates ML densest subgraph mining by 15 $\times$ on average and supports efficient FirmCore retrieval and temporal social network analysis, demonstrating its usefulness for real-world ML graph problems.

ACKNOWLEDGMENTS

This work was supported by the Aarhus University Research Foundation (grant 2022-0142).

REFERENCES

- [1] Akhlaque Ahmad, Lyuheng Yuan, Da Yan, Guimu Guo, Jieyang Chen, and Chengcui Zhang. 2023. Accelerating k-Core Decomposition by a GPU. In *ICDE 2023*. IEEE, 1818–1831.
- [2] Hidayet Aksu, Mustafa Canim, Yuan-Chi Chang, Ibrahim Korpeoglu, and Özgür Ulusoy. 2014. Distributed k-Core View Materialization and Maintenance for Large Dynamic Graphs. *TKDE 2014* 26, 10 (2014), 2439–2452.
- [3] José Ignacio Alvarez-Hamelin, Luca Dall'Asta, Alain Barrat, and Alessandro Vespignani. 2005. k-core decomposition: A tool for the visualization of large scale networks. *arXiv* (2005).
- [4] N Azimi-Tafreshi, J Gómez-Gardenes, and SN Dorogovtsev. 2014. k-Core percolation on multiplex networks. *Physical Review E* 90, 3 (2014), 032816.
- [5] Vladimir Batagelj and Matjaz Zaversnik. 2003. An o(m) algorithm for cores decomposition of networks. *arXiv* (2003).
- [6] Ali Behrouz and Farnoosh Hashemi. 2023. Generalized Densest Subgraph in Multiplex Networks. In *International Conference on Complex Networks and Their Applications*. Springer, 49–61.
- [7] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *WWW 2011*. ACM Press, 587–596.
- [8] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *WWW 2004*. ACM Press, Manhattan, USA, 595–601.
- [9] Francesco Bonchi, Francesco Gullo, Andreas Kaltenbrunner, and Yana Volkovich. 2014. Core decomposition of uncertain graphs. In *KDD 2014*. 1316–1325.
- [10] Fabio Celli, F Marta L Di Lascio, Matteo Magnani, Barbara Pacelli, and Luca Rossi. 2010. Social network data and practices: the case of friendfeed. In *SBP-BRIMS 2010*. Springer, 346–353.
- [11] Serhii Chalyi and Inna Pribylnova. 2019. The method of constructing recommendations online on the temporal dynamics of user interests using multilayer graph. *EUREKA: Physics and Engineering* 3 (2019), 13–19.
- [12] Moses Charikar. 2000. Greedy approximation algorithms for finding dense components in a graph. In *APPROX 2000*. Springer, 84–95.
- [13] Jie Chen and Yousef Saad. 2010. Dense subgraph extraction with application to community detection. *TKDE 2010* 24, 7 (2010), 1216–1230.
- [14] Shuntong Chen, Huaen Qian, Yanping Wu, Chen Chen, and Xiaoyang Wang. 2019. Efficient adoption maximization in multi-layer social networks. In *ICDMW 2019*. IEEE, 56–60.
- [15] Naga Shailaja Dasari, Ranjan Desh, and Mohammad Zubair. 2014. Park: An efficient algorithm for k-core decomposition on multicore processors. In *Big Data 2014*. IEEE, 9–16.
- [16] Manlio De Domenico, Antonio Lima, Paul Mougél, and Mirco Musolesi. 2013. The anatomy of a scientific rumor. *Scientific reports* 3, 1 (2013), 2980.
- [17] Manlio De Domenico, Vincenzo Nicosia, Alexandre Arenas, and Vito Latora. 2015. Structural reducibility of multilayer networks. *Nature communications* 6, 1 (2015), 6864.
- [18] Manlio De Domenico, Mason A Porter, and Alex Arenas. 2015. MuxViz: a tool for multilayer analysis and visualization of networks. *Journal of Complex Networks* 3, 2 (2015), 159–176.
- [19] Mark E Dickison, Matteo Magnani, and Luca Rossi. 2016. *Multilayer social networks*. Cambridge University Press.
- [20] Xiaoxi Du, Ruoming Jin, Liang Ding, Victor E Lee, and John H Thornton Jr. 2009. Migration motif: a spatial-temporal pattern mining approach for financial markets. In *KDD 2009*. 1135–1144.
- [21] David Eppstein, Maarten Löffler, and Darren Strash. 2010. Listing all maximal cliques in sparse graphs in near-optimal time. In *ISAAC 2010*. Springer, 403–414.
- [22] Eugene Fratkin, Brian T Naughton, Douglas L Brutlag, and Serafim Batzoglou. 2006. MotifCut: regulatory motifs finding with maximum density subgraphs. *Bioinformatics* 22, 14 (2006), e150–e157.
- [23] Edoardo Galimberti, Francesco Bonchi, and Francesco Gullo. 2017. Core decomposition and densest subgraph in multilayer networks. In *CIKM 2017*. 1807–1816.
- [24] Edoardo Galimberti, Francesco Bonchi, Francesco Gullo, and Tommaso Lanciano. 2020. Core decomposition in multilayer networks: Theory, algorithms, and applications. *TKDD 2020* 14, 1 (2020), 1–40.
- [25] Antonios Garas, Frank Schweitzer, and Shlomo Havlin. 2012. A k-shell decomposition method for weighted networks. *New Journal of Physics* 14, 8 (2012), 083030.
- [26] David Gibson, Ravi Kumar, and Andrew Tomkins. 2005. Discovering large dense subgraphs in massive graphs. In *VLDB 2005*. 721–732.
- [27] Neil Zhenqiang Gong, Wenchang Xu, Ling Huang, Prateek Mittal, Emil Stefanov, Vyas Sekar, and Dawn Song. 2012. Evolution of social-attribute networks: measurements, modeling, and implications using google+. In *Proceedings of the 2012 internet measurement conference*. 131–144.
- [28] Joseph L Greathouse and Mayank Daga. 2014. Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format. In *SC 2014*. IEEE, 769–780.
- [29] Farnoosh Hashemi, Ali Behrouz, and Laks VS Lakshmanan. 2022. Firmcore decomposition of multilayer networks. In *WWW 2022*. 1589–1600.
- [30] Petter Holme and Jari Saramäki. 2012. Temporal networks. *Physics reports* 519, 3 (2012), 97–125.
- [31] Haiyan Hu, Xifeng Yan, Yu Huang, Jiawei Han, and Xianghong Jasmine Zhou. 2005. Mining coherent dense subgraphs across massive biological networks for functional discovery. *Bioinformatics* 21, suppl_1 (2005), i213–i221.
- [32] Cheng Huang, Johannes Langguth, Xing Cai, Davide Mottin, and Ira Assent. 2026. Geld: Load-balanced D-Core Decomposition for Consumer GPUs. *Proceedings of the ACM on Management of Data* 4, 3 (SIGMOD (2026)), 1–26.
- [33] Joseph JáJá. 1992. *Parallel algorithms*.
- [34] Vinay Jethava and Niko Beerenwinkel. 2015. Finding dense subgraphs in relational graphs. In *ECML-PKDD 2015*. Springer, 641–654.
- [35] Humayun Kabir and Kamesh Madduri. 2017. Parallel k-core decomposition on multicore platforms. In *IPDPSW 2017*. IEEE, 1482–1491.
- [36] Samir Khuller and Barna Saha. 2009. On finding dense subgraphs. In *ICALP 2009*. Springer, 597–608.
- [37] Jérôme Kunegis. 2013. Konect: the koblenz network collection. In *WWW 2013*. 1343–1350.
- [38] Tommaso Lanciano, Atsushi Miyauchi, Adriano Fazzino, and Francesco Bonchi. 2024. A survey on the densest subgraph problem and its variants. *Comput. Surveys* 56, 8 (2024), 1–40.
- [39] Yutaka I Leon-Suematsu, Kentaro Inui, Sadao Kurohashi, and Yutaka Kidawara. 2011. Web spam detection by exploring densely connected subgraphs. In *WI-IAT 2011*, Vol. 1. IEEE, 124–129.
- [40] Jure Leskovec, Lada A Adamic, and Bernardo A Huberman. 2007. The dynamics of viral marketing. *TWEB 2007* 1, 1 (2007), 5–es.
- [41] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. <http://snap.stanford.edu/data>.
- [42] Don R Lick and Arthur T White. 1970. k-Degenerate graphs. *Canadian Journal of Mathematics* 22, 5 (1970), 1082–1096.
- [43] Boge Liu, Fan Zhang, Chen Zhang, Wenjie Zhang, and Xuemin Lin. 2019. Corecube: Core decomposition in multilayer graphs. In *WISE 2019*. Springer, 694–710.
- [44] Dandan Liu, Run-An Wang, Zhaonian Zou, and Xin HUANG. 2024. Fast Multilayer Core Decomposition and Indexing. In *ICDE 2024*.
- [45] Dandan Liu and Zhaonian Zou. 2023. Gcore: Exploring cross-layer cohesiveness in multi-layer graphs. *VLDB 2023* 16, 11 (2023), 3201–3213.
- [46] Xueming Liu, Enrico Maiorino, Arda Halu, Kimberly Glass, Rashmi B Prasad, Joseph Loscalzo, Jianxi Gao, and Amitabh Sharma. 2020. Robustness and lethality in multilayer biological molecular networks. *Nature communications* 11, 1 (2020), 6043.
- [47] Youzhe Liu, Xiaojun Dong, Yan Gu, and Yihan Sun. 2025. Parallel k-Core Decomposition: Theory and Practice. *Sigmod 2025* 3, 3 (2025). <https://doi.org/10.1145/3725332>
- [48] Zhicheng Liu and Jeffrey Heer. 2014. The effects of interactive latency on exploratory visual analysis. *IEEE transactions on visualization and computer graphics* 20, 12 (2014), 2122–2131.
- [49] Wensheng Luo, Chenhao Ma, Yixiang Fang, and Laks VS Lakshmanan. 2023. A survey of densest subgraph discovery on large graphs. *arXiv preprint arXiv:2306.07927* (2023).
- [50] Fragkiskos D Malliaros and Michalis Vazirgiannis. 2013. To stay or not to stay: modeling engagement dynamics in social graphs. In *CIKM 2013*. 469–478.
- [51] Alan Mislove, Hema Swetha Koppula, Krishna P Gummadri, Peter Druschel, and Bobby Bhattacharjee. 2008. Growth of the flickr social network. In *SIGCOMM Workshop*. 25–30.
- [52] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. 2011. Distributed k-core decomposition. In *SIGACT-SIGOPS 2011*. 207–208.
- [53] Elisa Omodei, Manlio De Domenico, and Alex Arenas. 2015. Characterizing interactions in online social networks during exceptional events. *Frontiers in Physics* 3 (2015), 59.
- [54] Katerina Pechlivanidou, Dimitrios Katsaros, and Leandros Tassioulas. 2014. MapReduce-based distributed k-shell decomposition for online social networks. In *SERVICES 2014*. IEEE, 30–37.
- [55] Ahmet Erdem Sariyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V Çatalyürek. 2013. Streaming algorithms for k-core decomposition. *VLDB 2013* 6, 6 (2013), 433–444.
- [56] Konstantinos Semertzidis, Evaggelia Pitoura, Evimaria Terzi, and Panayiotis Tsapras. 2019. Finding lasting dense subgraphs: K. Semertzidis et al. *Data mining and knowledge discovery* 33, 5 (2019), 1417–1445.
- [57] Kijung Shin, Tina Eliassi-Rad, and Christos Faloutsos. 2016. Corescope: Graph mining using k-core analysis—patterns, anomalies and algorithms. In *ICDM 2016*. IEEE, 469–478.
- [58] Kijung Shin, Tina Eliassi-Rad, and Christos Faloutsos. 2018. Patterns and anomalies in k-cores of real-world graphs with applications. *Knowledge and Information Systems* 54, 3 (2018), 677–710.
- [59] Run-An Wang, Zhaonian Zou, Dandan Liu, and Xudong Liu. 2025. FocusCores of Multilayer Graphs. *IEEE Transactions on Knowledge and Data Engineering* (2025).
- [60] Huanhuan Wu, James Cheng, Yi Lu, Yiping Ke, Yuzhen Huang, Da Yan, and Hejun Wu. 2015. Core decomposition in large temporal graphs. In *Big data 2015*.

- IEEE, 649–658.
- [61] Huihui Yang, Chunxue Zhu, Longlong Lin, and Pingpeng Yuan. 2024. Towards truss-based temporal community search. *arXiv preprint arXiv:2410.15046* (2024).
 - [62] Feng Yanghe and Wang Jiao Teng. 2013. Peeling the drug target proteins network by a core decomposition method. *Research Journal of Biotechnology* 8, 10 (2013), 66–70.
 - [63] Zin Mar Yin and Soe Soe Khaing. 2013. Multi-layered graph clustering in finding the community cores. *Int. J. Adv. Res. Comput. Eng. Technol* 2 (2013), 2674–2677.
 - [64] Songyang Zhang, Qinwen Deng, and Zhi Ding. 2023. Signal processing over multilayer graphs: Theoretical foundations and practical applications. *IEEE Internet of Things Journal* 11, 2 (2023), 2453–2471.
 - [65] Yikai Zhang, Jeffrey Xu Yu, Ying Zhang, and Lu Qin. 2017. A fast order-based approach for core maintenance. In *ICDE 2017*. IEEE, 337–348.
 - [66] Lihua Zhou, Jialong Wang, Yixin Song, Lizhen Wang, and Hongmei Chen. 2025. Community Search over Heterogeneous Information Networks: A Survey. *Comput. Surveys* 58, 4 (2025), 1–37.