Dupin: A Parallel Framework for Densest Subgraph Discovery in Fraud Detection on Massive Graphs

JIAXIN JIANG, National University of Singapore, Singapore
SIYUAN YAO, National University of Singapore, Singapore
YUCHEN LI, Singapore Management University, Singapore
QIANGE WANG, National University of Singapore, Singapore
BINGSHENG HE, National University of Singapore, Singapore
MIN CHEN, GrabTaxi Holdings, Singapore

Detecting fraudulent activities in financial and e-commerce transaction networks is crucial. One effective method for this is Densest Subgraph Discovery (DSD). However, deploying DSD methods in production systems faces substantial scalability challenges due to the predominantly sequential nature of existing methods, which impedes their ability to handle large-scale transaction networks and results in significant detection delays. To address these challenges, we introduce Dupin, a novel parallel processing framework designed for efficient DSD processing in billion-scale graphs. Dupin is powered by a processing engine that exploits the unique properties of the peeling process, with theoretical guarantees on detection quality and efficiency. Dupin provides user-friendly APIs for flexible customization of DSD objectives and ensures robust adaptability to diverse fraud detection scenarios. Empirical evaluations demonstrate that Dupin consistently outperforms several existing DSD methods, achieving performance improvements of up to 100 times compared to traditional approaches. On billion-scale graphs, Dupin demonstrates the potential to enhance the prevention of fraudulent transactions from 45.3% to 94.5% and reduces density error from 30.3% to below 5.0%, as supported by our experimental results. These findings highlight the effectiveness of Dupin in real-world applications, ensuring both speed and accuracy in fraud detection.

CCS Concepts: • **Theory of computation → Graph algorithms analysis**; **Approximation algorithms analysis**; **Massively parallel algorithms**.

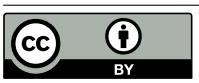
Additional Key Words and Phrases: Graph Algorithms, Parallel Programming, Densest Subgraph Discovery

ACM Reference Format:
Jiaxin Jiang, Siyuan Yao, Yuchen Li, Qiange Wang, Bingsheng He, and Min Chen. 2025. Dupin: A Parallel Framework for Densest Subgraph Discovery in Fraud Detection on Massive Graphs. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 150 (June 2025), 26 pages. https://doi.org/10.1145/3725287

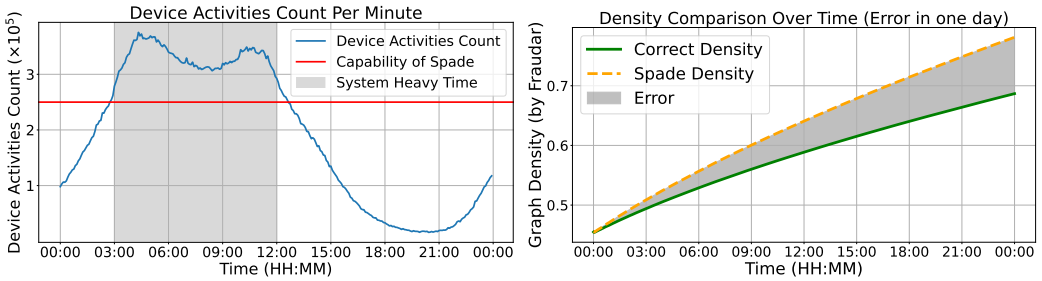
1 Introduction

In today's digital landscape, graph structures are essential in various applications, including transaction networks [10, 26, 52], messaging infrastructure [30, 37], and social networking platforms [19, 34, 51]. A challenge in these domains is detecting fraudulent activities, which can significantly undermine business integrity and consumer trust. The Densest Subgraph Discovery

Authors' Contact Information: Jiaxin Jiang, National University of Singapore, Singapore, jxjiang@nus.edu.sg; Siyuan Yao, National University of Singapore, Singapore, siyuan.y@u.nus.edu; Yuchen Li, Singapore Management University, Singapore, yuchenli@smu.edu.sg; Qiange Wang, National University of Singapore, Singapore, wang.qg@nus.edu.sg; Bingsheng He, National University of Singapore, Singapore, hebs@comp.nus.edu.sg; Min Chen, GrabTaxi Holdings, Singapore, min.chen@grab.com.



ACM 2836-6573/2025/6-ART150
https://doi.org/10.1145/3725287



(a) System Load Distribution. (b) Error Accumulation Trends.

Fig. 1. Activity analysis from our industry partner, Grab. Data normalized for privacy.

(DSD) problem, first explored by [22], has emerged as a critical tool in this context, facilitating tasks such as link spam identification [4, 20], community detection [9, 17], and notably, fraud detection [8, 24, 46]. The peeling process [2, 5, 8, 24, 50] is central to many DSD algorithms. This process removes vertices based on density metrics. *The density metrics are mainly differed by how the weight function is defined. They serve as indicators for the community's suspiciousness—denser subgraphs indicate more intense and cohesive interactions that are more likely to be fraudulent.* Examples of such metrics include vertex degree or the cumulative weight of adjacent edges. While these methods have demonstrated efficiency and robustness, their practical deployment in real-world fraud detection systems faces significant challenges. For instance, business requirements necessitate the detection of fraudulent communities within billion-scale graphs in mere seconds. Recent studies, including those by [1, 52], reveal that malicious bot activity in e-commerce accounted for a staggering 21.4% of all traffic in 2018, highlighting the urgency of developing effective detection mechanisms.

Existing Challenges. Over the years, extensive research has advanced DSD through incremental peeling algorithms [2, 18, 26, 27, 47] and parallel peeling algorithms [14, 16, 45, 46]. Despite these developments, practical fraud detection in real-world settings continues to face several limitations:

(1) **Lack of Flexibility in Density Metrics.** Many existing algorithms rely on fixed scoring functions tailored to DSD. Methods such as [4, 29] employ static detection criteria that become less effective as fraudulent behaviors diversify. In fraud detection, for instance, a density metric that incorporates transaction amounts as weights [23] highlights collusion patterns—such as frequent high-value or consistent transactions indicative of coupon or rebate abuse. Similarly, when detecting fraud communities, the density metric might utilize structural measures (*e.g.*, triangle counts [50]) to better capture community characteristics.

(2) **Inefficient Incremental Updates.** Current incremental algorithms often fail to scale to billion-edge graphs (G) with high transaction rates (*i.e.,* graph updates ΔG). For example, consider the state-of-the-art incremental algorithm Spade [27] (Figure 1a), where the vertical axis represents the Device Activities Count (i.e., the number of new edges generated per minute) and the gray area indicates System Heavy Time. Spade can process approximately 2.65×10^5 edge updates per minute, which is insufficient to handle scenarios with high transaction volumes. Moreover, these methods often assume static edge weights, even though newly inserted edges can alter the weights of existing edges. For example, the cumulative density error of [27] can exceed 15.0% in a single day (Figure 1b).

(3) **Ineffective Parallel Pruning.** Although parallelization can speed up DSD [45], existing parallel frameworks lack efficient pruning strategies, resulting in an excessive number of peeling iterations that degrade performance on large-scale datasets.

Industry Use Case. In collaboration with Grab, a leading technology company in Southeast Asia,

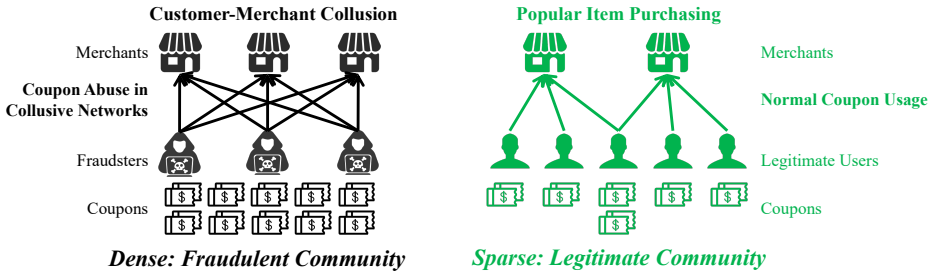


Fig. 2. Fraudulent coupon abuse vs. normal coupon usage in customer-merchant networks.

we address real-world fraud detection challenges on a digital payment and food delivery platform. Two predominant patterns are observed in practice (see Figure 2): (i) *Legitimate Community*, such as popular items attracting buyers, forms large but relatively sparse subgraphs. These communities reflect normal coupon usage and regular customer transactions across merchants. (ii) *Fraudulent Community*, on the other hand, are characterized by coupon abuse and collusion between customers and merchants. These involve smaller groups of participants engaging in frequent, repetitive transactions, resulting in highly dense subgraphs. The contrasting structures demonstrate the utility of density-based approaches for distinguishing legitimate transactions from fraudulent collusion. DSD algorithms then identify dense subgraphs indicative of fraudulent communities.

Latency and Prevention Ratio. Existing DSD approaches struggle to meet real-time latency targets and maintain high fraud-prevention ratios under heavy transaction loads:

- *Latency:* A representative incremental peeling approach is Spade [26, 27], which leverages existing peeling results on G and refines them incrementally when new vertices or edges (ΔG) arrive *to detect the dense subgraph on* $G \oplus \Delta G$. However, when ΔG is large, Spade incurs substantial overhead from frequent data structure updates and density recalculations. For instance, processing even a single batch of 1K edges on a 2-billion-edge graph (CPU Xeon X5650) can exceed 200 seconds, which is infeasible for real-time detection on modern e-commerce platforms that may see up to 400K edge insertions per minute.
- *Prevention Ratio:* Parallel DSD frameworks often compute dense subgraphs from scratch in parallel directly on $G \oplus \Delta G$. While this approach exploits multiple cores, it typically exhibits latency over 1000 seconds and suffers reduced detection effectiveness when scaling from million-edge to billion-edge datasets (e.g., dropping from 92.5% to 45.0% in fraud prevention).

To address these challenges, we pose the following research question: *How can we enhance the flexibility, efficiency, and scalability of* DSD *methods to effectively detect fraud in massive graphs?*

Contributions. We propose a DSD framework for fraud detection. Our key contributions are:

- **A** DSD **Parallelization Framework.** We propose Dupin, a novel parallel processing framework for Densest Subgraph Discovery (DSD) that breaks the sequential dependencies inherent in traditional density metrics, significantly enhancing scalability and efficiency on both weighted and unweighted graphs. Unlike existing methods that focus on specific density metrics, Dupin allows developers to define custom fraud detection semantics through flexible suspiciousness functions for edges and vertices. It supports a wide range of DSD semantics, including DG [6], DW [23], FD [24], TDS [50], and kCLiDS [14]. Dupin not only leverages parallelism but also provides theoretical guarantees on approximation ratios for the supported density metrics. To the best of our knowledge, this is the *first* generic parallel processing framework that accommodates a broad class of density metrics in DSD, offering both flexibility and theoretical assurance.
- **Exploiting Long-Tail Properties.** We identify and address the *long-tail* problem in the peeling process of DSD algorithms, where late iterations remove only a small fraction of vertices and

Table 1. Frequently used notations.

Notation	Definition
G / ΔG	a transaction graph / updates to graph G
a_i / c_{ij}	the suspiciousness on vertex u_i / on edge (u_i, u_j)
$N(u)$	the neighbors of vertex u
$g(S)$	the suspiciousness density of vertex set S
$f(S)$	the sum of the suspiciousness of vertex set S
$w_u(S)$	the decrease in f by peeling u from S, *i.e.,* the peeling weight
Q	the peeling algorithms
S^P	the vertex set returned by peeling algorithms
S^*	the optimal vertex set

contribute little to the quality of the densest subgraph, leading to inefficiency. To tackle this, we introduce two novel optimizations: *Global Pruning Optimization*, which proactively removes low-contribution vertices within each peeling iteration to reduce computational overhead, and *Local Pruning Optimization*, which eliminates disparate structures formed after vertex removal to enhance the density and quality of the detected subgraph. These optimizations significantly reduce the number of iterations required, improving both performance and result quality.

- **Empirical Validation and Real-World Impact:** We conduct comprehensive experiments on large-scale industry datasets to validate the effectiveness of Dupin. Our results show that Dupin achieves up to 100 times faster fraud detection compared to the state-of-the-art incremental method Spade [27] and parallel methods like PBBS [45]. Moreover, Dupin is capable of preventing up to 94.5% of potential fraud, demonstrating significant practical benefits and reinforcing system integrity in real-world applications.

Organization. The remainder of this paper is structured as follows: Section 2 presents the background and the literature review. Section 3 introduces the architecture of Dupin. In Section 4, we propose the theoretical foundations for parallel peeling-based algorithms for DSD. Section 5 introduces long-tail pruning techniques to improve efficiency and effectiveness. The experimental evaluation of Dupin is presented in Section 6. Finally, Section 7 concludes the paper and outlines avenues for future research.

2 Background and Related Work

2.1 Preliminary

This subsection introduces the preliminary concepts and notations. A summary of frequently used notations is provided in Table 1.

Graph G. In this work, we operate on a *weighted* graph denoted as $G = (V, E)$, where V represents the set of vertices and E ($\subseteq V \times V$) represents the set of edges. The graph can be either directed or undirected. Each edge $(u_i, u_j) \in E$ is assigned a *nonnegative* weight, expressed as c_{ij}. Additionally, $N(u)$ denotes the set of neighboring vertices connected to vertex u.

Induced Subgraph. Given a subset $S \subseteq V$, the *induced subgraph* is defined as $G[S] = (S, E[S])$, where $E[S] = \{(u, v) \in E \mid u, v \in S\}$. The size of the subset S is denoted by $|S|$.

Density Metrics g and Weight Function f. We utilize the class of metrics g as defined in prior works [6, 24, 27], where for a given subset of vertices $S \subseteq V$, the density metric is computed as: $g(S) = \frac{f(S)}{|S|}$. Here, $f(S)$ represents the total weight of the induced subgraph $G[S]$, which is defined as the sum of the weights of vertices in S and the weights of edges in $E[S]$. More formally:

$$f(S) = \sum_{u_i \in S} a_i + \sum_{(u_i, u_j) \in E[S]} c_{ij} \tag{1}$$

The weight of a vertex u_i, denoted as a_i ($a_i \geq 0$), quantifies the suspiciousness of user u_i. The weight of an edge (u_i, u_j), represented by c_{ij} ($c_{ij} > 0$), reflects the suspiciousness of the transaction between users u_i and u_j. Intuitively, the density metric $g(S)$ encapsulates the overall suspiciousness density of the induced subgraph $G[S]$. Larger $g(S)$ indicates a higher density of suspiciousness within $G[S]$. The vertex set that maximizes the density metric g is denoted by S^*.

We next introduce five representative density metrics employed in fraud detection applications.

Dense Subgraphs (DG) [6]. The DG metric is employed to quantify the connectivity of substructures within a graph. It has found extensive applications in various domains, such as detecting fake comments in online platforms [31] and identifying fraudulent activities in social networks [3]. Given a subset of vertices $S \subseteq V$, the weight function of DG is defined as $f(S) = |E[S]|$, where $|E[S]|$ denotes the number of edges in the induced subgraph $G[S]$.

Dense Subgraph on Weighted Graphs (DW) [23]. In transaction graphs, it is common to have weights assigned to the edges, representing quantities such as the transaction amount. The DW metric extends the concept of dense subgraphs to accommodate these weighted relationships. Given a subset of vertices $S \subseteq V$, the weight function of DW is defined as $f(S) = \sum_{(u_i,u_j)\in E[S]} c_{ij}$.

Fraudar (FD) [24]. In order to mitigate the effects of fraudulent actors deliberately obfuscating their activities, Hooi et al. [24] introduced the FD algorithm. This approach innovatively incorporates weights for edges and assigns prior suspiciousness scores to vertices, potentially utilizing side information to enhance detection accuracy. Given a vertex subset $S \subseteq V$, the weight function adopted by FD is defined as shown in Equation 1. In this formulation, a_i represents the suspiciousness score of vertex u_i, and the edge weight c_{ij} is given by $\frac{1}{\log(x+c)}$, where x denotes the degree of the object vertex between u_i and u_j, and c is a positive constant [24].

Triangle Densest Subgraph (TDS) [50]. Triangles serve as a crucial structural motif in graphs, offering insights into connectivity and cohesion. In TDS, the density of a subgraph $G[S]$ is measured by $f(S) = t(S)$, where $t(S)$ is the number of triangles in S. Although this directly counts triangles rather than vertices or edges, it can still be expressed in our general density framework $f(S) = \sum_{u_i\in S} a_i + \sum_{(u_i,u_j)\in E[S]} c_{ij}$ by assigning $c_{ij} = 0$ and $a_i = \frac{t_i}{3}$, where t_i is the number of triangles involving vertex u_i. Summing $\frac{t_i}{3}$ over all vertices in S counts each triangle exactly once, thus yielding $f(S) = t(S)$. Consequently, TDS fits naturally under the same density metric framework used for other weighted subgraphs.

k-Clique Densest Subgraph (kCLiDS) [14]. Extending TDS beyond triangles ($k = 3$) to any clique size k, kCLiDS defines the weight function as $f(S) = k(S)$, $k(S)$ is the number of k-cliques in $G[S]$. A k-Clique is a complete subgraph of k vertices, representing a tightly-knit group of vertices. Just as in the TDS case, kCLiDS can be embedded in our general framework by assigning $c_{ij} = 0$ for all edges and distributing each k-Clique's contribution among its k vertices. Specifically, let $a_i = k_i/k$, where k_i denotes the number of k-Cliques that include vertex u_i. Summing k_i/k over all $u_i \in S$ counts each k-clique exactly once, thereby recovering $f(S) = k(S)$ within the same density formulation.

Problem Statement. Given a graph $G = (V, E)$ and a community detection semantic with density metric $g(S)$, our goal is to find a vertex subset $S^* \subseteq V$ that maximizes $g(S^*)$.

2.2 Sequential Peeling Algorithms

Peeling algorithms have gained popularity for their efficiency, robustness, and proven worst-case performance in various applications [6, 24, 50]. Next, we provide a concise overview of the typical execution flow of these algorithms.

Peeling Weight. For a vertex set S, the *peeling weight* of a vertex $u_i \in S$, denoted by $w_{u_i}(S)$, is defined as the reduction in the value of the weight function f when u_i is removed from S.

Algorithm 1: Sequential Peeling Algorithm

Input: Graph $G = (V, E)$, density metric $g(S)$
Output: Subset S^* maximizing $g(S)$

1 $S_0 \leftarrow V, S^* \leftarrow S_0, i \leftarrow 1$ /* Initialization */
2 **while** $S_{i-1} \neq \emptyset$ **do**
3 $u_i \leftarrow \arg\min_{u \in S_{i-1}} w_u(S_{i-1})$ /* The vertex to be peeled */
4 $S_i \leftarrow S_{i-1} \setminus \{u_i\}$ /* Peel u_i from S_{i-1} */
5 **if** $g(S_i) > g(S^*)$ **then**
6 $S^* \leftarrow S_i$ /* Update optimal subset */
7 $i \leftarrow i + 1$
8 **return** S^* /* The subset that maximizes $g(S)$ */

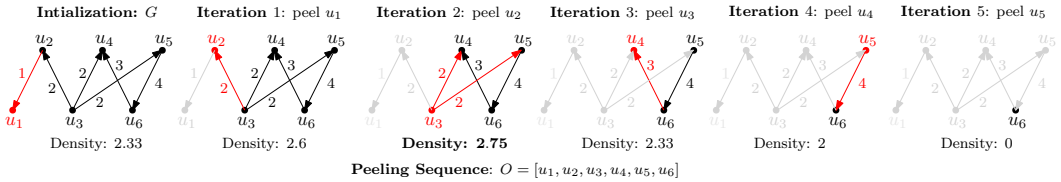


Fig. 3. Example of sequential peeling algorithms.

Sequential Peeling Algorithm (Algorithm 1). The peeling process starts with the full vertex set $S_0 = V$ and initializes the optimal subset S^* to S_0 (Line 1). The algorithm iterates as long as the current subset S_{i-1} is not empty (Line 3). In each iteration, the vertex u_i with the smallest peeling weight is identified and peeled from S_{i-1} to form S_i (Lines 3 and 4). If the density metric $g(S_i)$ exceeds that of the current optimal subset S^*, S^* is updated to S_i (Line 6). This process generates a sequence of nested subsets $S_0, S_1, \ldots, S_{|V|}$, ultimately returning the subset S^* that maximizes the density metric $g(S^*)$ (Line 8). The sequential peeling algorithm is a foundational approach widely adopted in existing studies [2, 8, 24, 26, 27, 50].

Approximation. An algorithm Q is defined as an α-approximation for the Densest Subgraph Discovery (DSD), where $\alpha \geq 1$, if for any given graph it returns a subset S satisfying the condition $g(S) \geq \frac{g(S^*)}{\alpha}$, where S^* is the optimal subset that maximizes the density metric g.

Theorem 2.1 ([24]). *For the vertex set S^p returned by Algorithm 1 and the optimal vertex set S^*, it holds that $g(S^p) \geq \frac{g(S^*)}{2}$ for* DG, DW *and* FD *as the density metrics.*

The proofs for the theorems presented in this section are provided in Appendix B of [28].

Example 2.1 (Sequential Peeling Algorithm Process). *Consider the graph G as shown in Figure 3. The process begins with an initial graph density of 2.33. In the first iteration, vertex u_1 is peeled due to its smallest peeling weight. Subsequently, vertices are peeled in the following order: u_2, u_3, u_4, u_5, and u_6 based on the updated criteria after each peeling operation. The sequence of peeling continues until the last vertex, u_6, is peeled, resulting in a final graph density of 0. The peeling sequence, therefore, is $O = [u_1, u_2, u_3, u_4, u_5, u_6]$, completely disassembling the graph. Notably, after peeling u_2, the graph density increases to 2.75, reaching its maximum before beginning to decrease. This non-monotonic behavior is characteristic of the peeling process. The induced subgraph $G[S]$ of $S = [u_3, u_4, u_5, u_6]$ has the greatest density of 2.75, thus $G[S]$ is returned.*

Theorem 2.2 extends the analysis to TDS and kCLiDS, proof is omitted due to space limitation.

Theorem 2.2 ([50]). *For the vertex set S^p returned by Algorithm 1 and the optimal vertex set S^*, it holds that $g(S^p) \geq \frac{g(S^*)}{k}$ for* TDS *and k*CLiDS *as the density metrics where $k = 3$ for* TDS.

Table 2. Comparison of Algorithms Across Key Dimensions.

Systems[1]	Density Metric Support	Parallelizability	Weighted Graph	Pruning
Spade	DG, DW, FD, TDS, kCLiDS	Sequential	Yes	No
GBBS*	DG, DW, FD	Parallel	No	No
PKMC*	DG, DW, FD	Parallel	No	No
FWA*	DG, DW, FD	Parallel	No	No
ALENEX*	DG, DW, FD	Parallel	No	No
kCLIST	TDS, kCLiDS	Parallel	No	No
PBBS	TDS, kCLiDS	Parallel	No	No
Dupin	DG, DW, FD, TDS, kCLiDS	Parallel	Yes	Yes

The sequential peeling algorithm is designed with a 2-approximation guarantee; that is, when using density metrics such as DG, DW, and FD to evaluate suspiciousness, the density (and hence the suspiciousness) of the returned subgraph is at least half that of the optimal subgraph—in other words, the optimal subgraph's suspiciousness cannot exceed twice that of the sequential algorithm's output. In contrast, when the density metrics TDS and kCLiDS (which are based on triangle counts and k-Clique counts, respectively) are employed as indicators of suspiciousness, the sequential peeling algorithm guarantees a 3-approximation. Thus, the suspiciousness of the returned subgraph is within a factor of 3 of the optimal value.

2.3 Related Work

Densest Subgraph Discovery (DSD). DSD represents a core area of research in graph theory, extensively explored in literature [7, 8, 21, 32, 43, 44]. The foundational max-flow-based algorithm by Goldberg et al. [22] set a benchmark for exact DSD, with subsequent studies introducing more scalable exact solutions [36, 38]. While speed enhancements have been achieved through various greedy approaches [6, 8, 24, 36], these methods are inherently sequential and difficult to parallelize. For example, they often peel one vertex at a time, which results in too many peeling iterations. This sequential nature limits their scalability and efficiency on massive graphs. In graph-based fraud detection, techniques like [4] and [29] leverage local search heuristics to identify dense subgraphs in bipartite networks, frequently applied in fraud, spam, and community detection across social and review platforms [24, 41, 46]. Despite their widespread use, they suffer from efficiency issues in large graphs, often involving numerous iterations and lacking effective pruning techniques, exacerbating the long-tail problem. Moreover, they are limited to specific definitions of dense subgraphs. In contrast, Dupin offers a versatile, parallel DSD approach. It facilitates the simultaneous peeling of multiple vertices, reducing the computational overhead. Notably, Dupin introduces unique global and local pruning techniques that enhance efficiency, allowing it to handle large-scale graphs effectively. Moreover, Dupin provides APIs that allow users to customize definitions of dense subgraphs, ensuring that the resultant graph density adheres to theoretical bounds. This flexibility and efficiency make Dupin a significant advancement over existing methods.

DSD on Dynamic Graphs. To tackle real-time fraud detection, several variants have been designed for dynamic graphs [10, 18, 26, 27, 47]. [27] addresses real-time fraud community detection on evolving graphs (edge insertions) using an incremental peeling sequence reordering method, while [26] extends this approach to fully dynamic graphs (edge insertions and deletions). [47] proposes an incremental algorithm for maintaining and updating a dense subtensor over a tensor stream. [12] and [40] utilize sliding windows to detect dense subgraphs and bursting cores within specific time windows. These methods have three limitations. First, response time is an issue. Although

[1]Entries marked with (*) indicate methods that support only a single density metric and require modifications to the codebase for more metrics.

Fig. 4. Architecture of Dupin.

they respond quickly to benign communities, the results can change drastically for newly formed fraudulent communities, causing significant delays in incremental calculations for fraud detection. Second, efficiency is problematic. While incremental processing is faster than recomputation for a single transaction on billion-scale graphs, frequent incremental evaluations can be slower than a full recomputation. Third, in terms of flexibility, most methods support only a single semantic. In contrast, Dupin introduces a unified parallel framework for peeling-based dense subgraph detection. Dupin can respond within seconds on billion-scale graphs and defines multiple density metrics that can be used on the Dupin platform addressing the limitations of existing dynamic graph methods.

Parallel DSD**.** Several parallel methods for DSD have been proposed [2, 14, 16, 43, 45, 46]. Bahmani et al. [2] proposed a parallel algorithm for DG using the MapReduce framework. FWA [15] proposed a parallel algorithm for DW based on the Frank-Wolfe algorithm [25] using OpenMP [13]. Some works propose parallel k-core algorithms [35, 45, 48]. Others introduce parallel algorithms for k-Clique detection, with approaches offering a $(1+\epsilon)$-approximation ratio and parallel implementations for the kCLiDS problem [16, 49]. There are also algorithms for k-Clique densest subgraph detection using OpenMP [14], and methods for dense subtensor detection [46]. However, these systems have two main limitations. First, they typically support only a single-density metric and do not handle weighted graphs. In real-world applications, transactions between users can have varying degrees of importance based on amounts, frequencies [27], and other factors. Extending these systems to support other dense metrics with theoretical guarantees is non-trivial. Second, they often lack efficient pruning techniques, requiring numerous iterations and suffering from the long-tail issue, which hinders scalability on massive graphs. In contrast, Dupin defines the characteristics of a set of supported dense metrics and accommodates weighted subgraphs, allowing for more nuanced analysis reflective of real-world data. Our method introduces efficient pruning strategies by peeling multiple vertices in parallel based on a threshold mechanism, significantly reducing the number of iterations and mitigating the long-tail problem. This makes Dupin more effective and scalable in diverse real-world scenarios compared to previous approaches.

3 Architecture of Dupin

Parallelization offers a viable solution to these challenges by distributing the computational load across multiple processors. Modern CPUs are equipped with numerous cores that are cost-effective, enabling parallel processing. If the peeling process of DSD is parallelized, different processors can simultaneously evaluate and peel different nodes, thereby increasing scalability and efficiency. To enable users to design complex DSD algorithms to detect fraudsters based on our framework, Dupin, we have introduced the architecture, APIs, and supported density metrics in this section.

Architecture of Dupin **(Figure 4).** Developers can define custom fraud detection semantics by leveraging APIs to specify density metrics like DG, DW, FD, TDS, and kCLiDS. Dupin begins with graph loading and processes the input through the Dupin computing engine, which incorporates parallel peeling algorithms for scalable processing. To further optimize performance, both global and local peeling optimizations are implemented, enabling faster convergence and improved subgraph density at each iteration. The identified fraudulent communities are flagged and forwarded to

moderators for detailed analysis and appropriate actions such as banning suspicious entities. This modular design ensures extensibility and adaptability for various graph-based fraud detection tasks.

```
class Dupin {
public:
    Graph LoadGraph(const std::string& path); // Load graph from file
    // Set vertex suspiciousness function
    void VSusp(std::function<double(const Vertex& v, const Graph& g)> susp);
    // Set edge suspiciousness function
    void ESusp(std::function<double(const Edge&, const Graph&)> susp);
    void setEpsilon(double epsilon);
    void setK(int k);
    // Detect fraudsters using parallel processing
    std::vector<Vertex> ParDetect();
private:
    Graph _graph;  // Transaction graph
    std::vector<Vertex> _peelingSequence;
    std::vector<double> _peelingWeights;
    double _epsilon; }
```

Listing 1. APIs of Dupin.

APIs of Dupin **(Listing 1).** Similar to Spade, Dupin simplifies the definition of fraud detection semantics by allowing users to specify a small set of intuitive functions, such as VSusp and ESusp. In addition, Dupin provides a richer set of APIs that help users effectively balance detection speed and accuracy effectively. The key APIs encompass several components designed to facilitate the customization and optimization of fraud detection processes. These components provide a robust framework for users to implement their fraud detection strategies, ensuring both high performance and precision in identifying fraudulent activities. Details are listed below:

- VSusp and ESusp. Given a vertex or an edge, these components compute its suspiciousness based on user-defined strategies.
- isBenign. This component is used to decide whether a vertex is benign during the whole peeling process (Section 5). If the vertex is benign, it is peeled within the current iteration.
- setEpsilon. This function is utilized to control the precision of fraud detection. During periods of high system load, moderators can increase ϵ to achieve higher throughput. Conversely, when the system load is low, ϵ can be decreased to enhance accuracy.
- setK. This function defines the k-Clique parameter for analysis.

Implementation Example (Listing 2). To implement FD [24] on Dupin, users simply specify the vertex suspiciousness function vsusp by calling VSusp, and the edge suspiciousness function esusp by calling ESusp. Specifically: (1) vsusp is a constant function, i.e., for a vertex u_i, $\text{vsusp}(u_i) = a_i$; and (2) esusp is a logarithmic function such that, for an edge (u_i, u_j), $\text{esusp}(u_i, u_j) = \frac{1}{\log(x+c)}$, where x denotes the degree of the object vertex between u_i and u_j, and c is a positive constant [24]. Dupin enables developers to implement customized peeling algorithms, significantly reducing engineering overhead. The implementation of other metrics, including DG, DW, TDS, and kCLiDS, follows a similar procedure. Detailed implementations of these metrics are provided in Appendix B of [28].

```
1 double vsusp(const Vertex& v, const Graph& g) {
2   return g.weight[v]; } // Side information on vertex
3 double esusp(const Edge& e, const Graph& g) {
4   return 1.0 / log(g.deg[e.src] + 5.0); }
5 int main() {
6   Dupin dupin;
7   dupin.VSusp(vsusp); // Plug in vsusp
8   dupin.ESusp(esusp); // Plug in esusp
9   dupin.setEpsilon(0.1);
10  dupin.LoadGraph("graph_sample_path");
11  vector<Vertex> fraudsters = dupin.ParDetect();
12  return 0; }
```

Listing 2. Implementation of FD on Dupin.

Density Metrics in Dupin**.** To demonstrate the flexibility of Dupin in accommodating various density metrics, we establish comprehensive sufficient conditions outlined in Property 3.1. These conditions ensure that, for any density metric $g(S)$ satisfying the specified criteria, Dupin can parallelize the peeling process with theoretical guarantees. This enables Dupin to integrate a diverse array of metrics for effectively assessing graph density, which is essential for sophisticated fraud detection and network analysis tasks.

Property 3.1. *Given a density metric $g(S)$, if the following conditions are satisfied, then* Dupin *can parallelize the peeling process with theoretical guarantees:*

(1) $g(S) = \frac{f(S)}{|S|}$, *where f is a monotone increasing function.*
(2) $a_i \geq 0$, *ensuring non-negative vertex weight function.*
(3) $c_{ij} \geq 0$, *ensuring non-negative edge weight function.*

4 Parallelable Peeling Algorithms

For sequential peeling algorithms, only one vertex can be removed at a time, with each vertex's removal depending on the removal of previous vertices. This dependency prevents the algorithm from executing multiple vertex removals simultaneously.

In this section, we introduce a parallel peeling paradigm to break this dependency chain. This new approach peels vertices in parallel without affecting the programming abstractions that Dupin provides to end users. We then demonstrate how this paradigm offers fine-grained trade-offs between parallelism and approximation guarantees for all density metrics studied in this work.

4.1 Parallel Peeling Paradigm

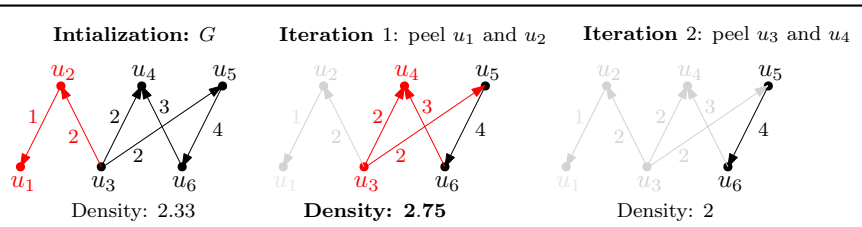
Instead of peeling the vertex with the maximum peeling weight, we can peel all vertices that lead to a significant decrease in the density scores of the remaining graph. This decrease is controlled by a tunable parameter ϵ. A larger ϵ allows more vertices to be peeled in parallel, while a smaller ϵ provides a better approximation ratio.

Parallel Peeling (Algorithm 2). Let S_i denote the vertex set after the i-th peeling iteration. Initially, $S_0 = V$ (Line 1). In each iteration, the algorithm computes vertex and edge suspiciousness in parallel, then aggregates the total weight $f(S_{i-1})$ using a *parallel* summation. Based on the density, the algorithm concurrently evaluates each vertex $u \in S_{i-1}$ to determine if its peeling weight $w_u(S_{i-1})$ satisfies $w_u(S_{i-1}) \leq k(1+\epsilon)g(S_{i-1})$. Vertices meeting this condition are simultaneously added to the removal set U. The algorithm then removes all vertices in U from S_{i-1}. Parameter k is determined by the metric used. $k = 2$ works for DG, DW and FD. $k = 3$ works for TDS and k is the size of the

Algorithm 2: Dupin: Parallel Peeling

Input: $G = (V, E)$, a density metric $g(S)$, and $\epsilon \geq 0$
Output: A subset of vertices S that maximizes the density metric $g(S)$

$S_0 \leftarrow V$ /* Initialize the subset with all vertices */
$i \leftarrow 1$ /* Initialize the index */
while $S_{i-1} \neq \emptyset$ **do**
  parallel_for $u_j \in S_{i-1}$ **do**
    $a_j \leftarrow \text{vsusp}(u_j)$
  parallel_for $(u_j, u_k) \in E$ ***and*** $u_j, u_k \in S_{i-1}$ **do**
    $c_{jk} \leftarrow \text{esusp}(u_j, u_k)$
  $f(S_{i-1}) \leftarrow \texttt{parallel\_sum}(a_j) + \texttt{parallel\_sum}(c_{jk})$
  $g(S_{i-1}) \leftarrow f(S_{i-1}) / |S_{i-1}|$
  $\tau \leftarrow k(1+\epsilon) \cdot g(S_{i-1})$ /* The threshold for peeling */
  $U \leftarrow \emptyset$ /* Initialize the removal set */
  parallel_for $u \in S_{i-1}$ **do**
    if $w_u(S_{i-1}) \leq \tau$ **then**
      $U \leftarrow U \cup \{u\}$ /* Vertices to be peeled */
  $S_i \leftarrow S_{i-1} \setminus U$ /* Update the subset */
  $i \leftarrow i + 1$ /* Increment the index */
return $\arg\max_{S_i} g(S_i)$ /* The subset that maximizes $g(S)$ */



Peeling Sequence: $O = [u_1, u_2; u_3, u_4; u_5, u_6]$

Fig. 5. Illustration of Parallel DW. Nodes marked in red indicate those to be peeled in the current iteration. The grayed areas represent nodes that have been peeled.

cliques for kCLiDS. This process is repeated recursively until no vertices are left, resulting in a series of vertex sets $S_0, \ldots, S_R$, where R is the number of iterations. The algorithm then returns the set S_i ($i \in [0, R]$) that maximizes $g(S_i)$, denoted as S^p.

In the following two examples, DW and kCLiDS are the density metrics for DSD, respectively.

Example 4.1 (Parallel Peeling Algorithm). *Consider the graph G as illustrated in Figure 5. The parallel peeling algorithm significantly reduces the number of iterations needed.*

(1) In the first iteration, the peeling weights of all vertices are computed concurrently. Vertices u_1 and u_2 have peeling weights of 1 and 3, respectively—both of which are below twice the current density—so they satisfy the peeling criteria and are removed simultaneously. This results in an updated graph density of 2.75.

(2) The second iteration sees the simultaneous peeling of u_3 and u_4, leaving the remaining vertices u_5 and u_6 disconnected and thus effectively peeled from the graph as well.

The parallel algorithm completes the peeling process in only two iterations, a notable improvement over sequential methods. Consequently, the peeling sequence is represented as $O = [u_1, u_2; u_3, u_4; u_5, u_6]$, where semicolons denote the parallel groups peeled in each iteration. The peeling process is non-monotonic; the induced subgraph $G[S]$ with $S = \{u_3, u_4, u_5, u_6\}$ has the maximum density 2.75 and is returned.

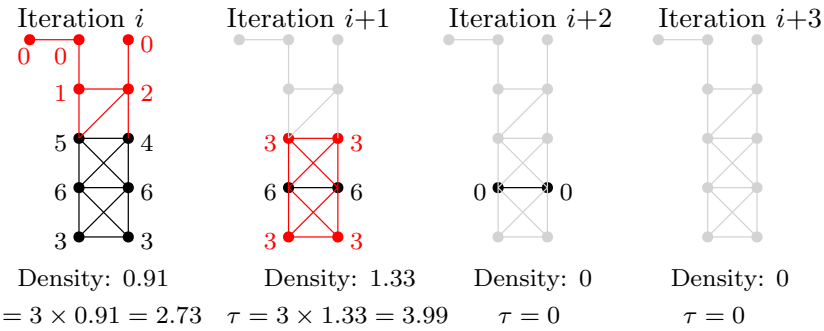


Fig. 6. Illustration of Parallel kCLiDS Peeling for $k = 3$. Labels adjacent to each node denote the count of k-Cliques that the node participates in within the current subgraph.

Example 4.2. *In Figure 6, we demonstrate the prowess of* Dupin *in executing parallel node peeling for* TDS *with $k = 3$. Consider the i-th iteration illustrated on the left: the five vertices in red would require five iterations to peel using sequential algorithms.*
Prior methodologies, such as those described in [45], could necessitate up to three iterations for complete peeling, as they are constrained to peeling vertices with identical weights concurrently and the recalculations of peeling weights are time-intensive.
In contrast, Dupin *computes, in parallel, the number of k-Cliques each vertex participates in and checks whether this count falls below the peeling threshold. This parallel evaluation enables the simultaneous removal of all five vertices in a single iteration, thereby producing a denser subgraph for the subsequent iteration. Overall,* Dupin *reduces the total peeling process to three iterations to extract the target dense subgraph, highlighting its superior efficiency in leveraging parallelism.*

4.2 Theoretical Analysis

In this section, we analyze the theoretical properties of the parallel peeling process. We begin by examining the number of iterations required for parallel peeling. Built upon the result, we then determine the time complexity and the approximation ratio of the parallel peeling process.

Lemma 4.1. *Let R denote the upper bound on the number of rounds required for the peeling process. For any $\epsilon > 0$, the peeling process requires at most $R < \log_{1+\epsilon}|V|$ rounds.*

Proof. We prove that for any $\epsilon \geq 0$, the size of the remaining vertex set is bounded by $|S_i| < \frac{1}{1+\epsilon}|S_{i-1}|$ for all $i \in [1, R]$. We first examine the peeling weights of the vertices in S_{i-1} for the density metrics DG, DW and FD where $k = 2$. As each edge weight in the peeling weight calculation contributes twice (once for each endpoint), and the vertex weights contribute once, we have

$$\sum_{u \in S_{i-1}} w_u(S_{i-1}) \leq 2f(S_{i-1}), \tag{2}$$

The peeling weights of the vertices in S_{i-1} are calculated as follows:

$$\sum_{u \in S_{i-1}} w_u(S_{i-1}) = \sum_{u \in S_{i-1} \setminus S_i} w_u(S_{i-1}) + \sum_{u \in S_i} w_u(S_{i-1}) \geq \sum_{u \in S_i} w_u(S_{i-1}) > 2(1+\epsilon)|S_i|g(S_{i-1}) \tag{3}$$

Combining Equation 2 and Equation 3, we have:

$$2(1+\epsilon)|S_i|g(S_{i-1}) < 2f(S_{i-1}) = 2|S_{i-1}|g(S_{i-1}) \tag{4}$$

For all $i \in [1, R]$,

$$|S_i| < \frac{1}{1+\epsilon}|S_{i-1}| \tag{5}$$

$$\Rightarrow |V| = |S_0| > \frac{1}{1+\epsilon}|S_1| > \ldots > (\frac{1}{1+\epsilon})^R |S_R| \tag{6}$$

Therefore, we have $R < \log_{1+\epsilon}|V|$.

For the density metrics TDS and kCLiDS, we use the fact that each edge contributes to the peeling weight calculation exactly k times due to the counting of each edge within every k-Clique in the graph ($k = 3$ for TDS). This multiplicity of edge contributions leads to the relationship:

$$\sum_{u \in S_{i-1}} w_u(S_{i-1}) \leq k f(S_{i-1}), \tag{7}$$

We then adapt Equations 3-6 and the proof follows. □

Time Complexity. Algorithm 2 operates for at most $R = \log_{1+\epsilon}|V|$ iterations, with each iteration scanning all vertices, taking $O(|V|)$ time. As vertices are peeled, updates are necessitated for the peeling weights of their neighboring vertices. Over the course of R iterations, there are no more than $|E|$ such updates for DG, DW and FD. Consequently, Algorithm 3 incurs $O((|E|+|V|)\log_{1+\epsilon}|V|)$ node/edge weight updates. For DG, DW and FD, the update costs are $O(1)$. For TDS and kCLiDS, we follow previous studies [11, 14] to compute the peeling weights, and the complexity is $O(k|E|\alpha(G)^{k-2})$, where $\alpha(G)$ is the arboricity of the graph. Hence, the overall complexity is $O(k|E|\alpha(G)^{k-2}(|E|+|V|)\log_{1+\epsilon}|V|)$. The cost of peeling weight updates is orthogonal to the Dupin framework.

Approximation Ratio. Next, we are ready to show that Dupin provides a theoretical guarantee with a $k(1+\epsilon)$-approximation for all density metrics studied in this work. We denote $u_i \in S^p$ as the first vertex in S^* peeled by Dupin, and S' as the peeling subsequence starting from u_i.

Theorem 4.2. *For S^p returned by* Dupin *and the optimal vertex set S^*, $g(S^p) \geq \frac{g(S^*)}{k(1+\epsilon)}$.*

Proof. We first prove that $\forall u_i \in S^*$, $w_{u_i}(S^*) \geq g(S^*)$. Suppose $w_{u_i}(S^*) < g(S^*)$. By peeling u_i from S^*, we have the following.

$$g(S^* \setminus \{u_i\}) = \frac{f(S^*) - w_{u_i}(S^*)}{|S^*|-1} > \frac{f(S^*) - g(S_i)}{|S^*|-1} = \frac{f(S^*) - g(S^*)}{|S^*|-1} = \frac{f(S^*) - \frac{f(S^*)}{|S^*|}}{|S^*|-1} = g(S^*) \tag{8}$$

Therefore, removing u_i from S^* would yield a better solution, contradicting the assumption that S^* is optimal. We can conclude that $w_{u_i}(S^*) \geq g(S^*)$.

Next, we assume that $u_i \in S^p$ is the first vertex in S^* peeled by the peeling algorithm. Since S^* is the optimal vertex set, we have

$$g(S^*) \leq w_{u_i}(S^*), \tag{9}$$

because the optimal value $g(S^*)$ must be less than or equal to the weight contribution of any single vertex in S^*, in particular u_i. Since $S^* \subseteq S'$, it is clear that

$$w_{u_i}(S^*) \leq w_{u_i}(S') \tag{10}$$

By the peeling condition, we have

$$w_{u_i}(S') \leq k(1+\epsilon)g(S') \tag{11}$$

$$\Rightarrow g(S^*) \leq w_{u_i}(S^*) \leq w_{u_i}(S') \leq k(1+\epsilon)g(S') \leq k(1+\epsilon)g(S^p) \tag{12}$$

Hence, we can conclude that $g(S^p) \geq \frac{g(S^*)}{k(1+\epsilon)}$. □

Summary. Dupin can balance efficiency and the worst-case approximation ratio with a single parameter, ϵ. A larger ϵ allows more vertices to be peeled in parallel, reducing peeling iterations. Meanwhile, the worst-case approximation ratio of parallel peeling is only affected by a factor of $(1+\epsilon)$ compared to the ratio for sequential peeling *w.r.t.* all density metrics studied in this work.

5 Long-Tail Pruning

We observe that the parallel peeling algorithm encounters two significant issues. First, there is a long-tail problem where subsequent iterations fail to produce denser subgraphs. Instead, each iteration only peels off a small fraction of vertices, which significantly impacts the efficiency. Second, during batch peeling, each peeling iteration often results in disparate structures. These are primarily caused by the peeling of certain vertices, which leaves their neighboring structure disconnected and fragmented. This phenomenon affects the continuity and coherence of the subgraphs.

As shown in Table 3, on the `la` dataset (the largest social network dataset tested in our experiments), DG, DW, and FD experience 24.7%, 46.8%, and 3.0% of rounds as ineffective long-tail iterations, severely limiting response time. Additionally, during the peeling process, if the batch size is large (e.g., $\epsilon = 0.5$), approximately 25.3%, 29.5%, and 7.2% of nodes become disconnected and sparse, significantly impacting the quality of the detected subgraphs.

Table 3. Impact of GPO and LPO on Peeling Rounds for Various Metrics. Experimented on dataset `la`.

Method	DG	DW	FD
Rounds without GPO	17,637	150,223	112,074
Rounds with GPO	13,287	79,835	108,706
Long-tail vertices	45,017,232	48,248,685	5,658,425
% Reduction in Rounds	24.7%	46.8%	3.0%
Rounds with LPO	3,221	10,832	101,255
Sparse vertices	13,324,405	15,487,382	3,762,288
% Reduction in Rounds	81.7%	92.8%	9.7%

To address the challenges of high iteration counts in peeling algorithms, Dupin introduces two long-tail pruning techniques: Global Peeling Optimization (GPO) to maintain a global peeling threshold, and Local Peeling Optimization (LPO) to improve the density of the current peeling iteration's subgraph. We tested three algorithms, DG, DW, and FD, and found that they required 17,637, 150,223, and 112,074 iterations respectively, which is significantly high. By strategically pruning long-tail vertices and sparse vertices, the number of iterations can be reduced by up to 46.8%. Additionally, LPO can further reduce the number of iterations by up to 92.8%.

5.1 Global Peeling Optimization

Initially, we define what constitutes a *long-tail vertex* during the peeling process. Peeling these long-tail vertices results in ineffective peeling iterations. To address this, we maintain a global maximum peeling threshold to determine which vertices can be peeled directly. When the global peeling threshold is greater than the current iteration's threshold, we use the global peeling threshold.

Definition 5.1 (Long-Tail vertex). *Given an α-approximation* DSD *algorithm and a set of vertices $S \subseteq V$, if $w_u(S) < \frac{g(S^*)}{\alpha}$ and S^* is the optimal vertex set, then u is a long-tail vertex.*

Lemma 5.1. *If $\exists u \in S_i$ is a long-tail vertex, then $S_i \neq S^P$.*

Proof. Assume that $S_i = S^P$. Since u is a long-tail vertex in S_i, $w_u(S_i) < \frac{g(S^*)}{\alpha} < g(S^P)$. By peeling u from S^P, we have the following:

$$g(S^P \setminus \{u\}) = \frac{f(S^P) - w_u(S^P)}{|S^P|-1} > \frac{f(S^P) - g(S^P)}{|S^P|-1} = \frac{f(S^P) - \frac{f(S^P)}{|S^P|}}{|S^P|-1} = g(S^P) \tag{13}$$

Peeling u from S^P yields a denser subgraph, contradicting that $S_i = S^P$. Hence, $S_i \neq S^P$. □

Algorithm 3: DupinGPO: Global Peeling Optimization

Input: $G = (V, E)$, a density metric $g(S)$, and $\epsilon \geq 0$
Output: A subset of vertices S^p that maximizes the density metric $g(S)$

1 $S_0 \leftarrow V$ /* Initialize the subset with all vertices */
2 $i \leftarrow 1$ /* Initialize the index */
3 $\tau_{\max} \leftarrow 0$ /* Initialize the global threshold */
4 **while** $S_{i-1} \neq \emptyset$ **do**
5 **parallel_for** $u_j \in S_{i-1}$ **do**
6 $a_j \leftarrow \text{vsusp}(u_j)$
7 **parallel_for** $(u_j, u_k) \in E$ ***and*** $u_j, u_k \in S_{i-1}$ **do**
8 $c_{jk} \leftarrow \text{esusp}(u_j, u_k)$
9 $f(S_{i-1}) \leftarrow \texttt{parallel\_sum}(a_j) + \texttt{parallel\_sum}(c_{jk})$
10 $g(S_{i-1}) \leftarrow f(S_{i-1}) / |S_{i-1}|$
11 $\tau_{\max} \leftarrow \max\{\tau_{\max}, \frac{g(S_{i-1})}{k(1+\epsilon)}\}$ /* Refine the global threshold */
12 $\tau \leftarrow \max\{\tau_{\max}, k(1+\epsilon) \cdot g(S_{i-1})\}$ /* Peeling threshold */
13 $U \leftarrow \emptyset$ /* Initialize the removal set */
14 **parallel_for** $u \in S_{i-1}$ **do**
15 **if** $w_u(S_{i-1}) \leq \tau$ **then**
16 $U \leftarrow U \cup \{u\}$ /* Vertices to be peeled */
17 $S_i \leftarrow S_{i-1} \setminus U$ /* Update the subset */
18 $i \leftarrow i + 1$ /* Increment the index */
19 **return** $\arg\max_{S_i} g(S_i)$ /* The subset that maximizes $g(S)$ */

Identifying Long-Tail Vertices. Based on Lemma 5.2, we can also establish that a vertex u in set S can be classified as a long-tail vertex if $w_u(S) < \frac{g(S^*)}{k(1+\epsilon)}$. This allows us to implement a global peeling threshold, denoted as $\tau_{\max}$. During each iteration, we compare $\frac{g(S_i)}{k(1+\epsilon)}$ with $\tau_{\max}$. If the former exceeds the latter, Dupin updates $\tau_{\max}$ to $\frac{g(S_i)}{k(1+\epsilon)}$, thereby enhancing the efficiency of the peeling.

Peeling with Global Threshold (Algorithm 3). Using FD as an example, we can set k to 2. In Lemma 5.2, we recognize that long-tail vertices in set S_i can be peeled directly during iteration i for FD. To facilitate this, Dupin initiates with a global peeling threshold $\tau_{\max}$ (Line 3). After identifying a denser subgraph post-peeling, $\tau_{\max}$ is updated to $\frac{g(S_i)}{2(1+\epsilon)}$, refining the peeling process (Line 11).

Example 5.1. *In the initial depiction provided in the leftmost panel of Figure 7 (the i-th iteration), the graph's density is* 4.28. *Traditionally, this scenario would need two additional iterations to complete the peeling process, which could increase computational time. As illustrated, vertex u cannot be peeled in the $(i+1)$-th iteration and require an extra iteration for its removal. However,* Dupin *can promptly peel vertices that fall below our algorithm's global peeling threshold, which is calculated as half of the current density ($\tau_{\max} = 4.28/2 = 2.14$).*
This approach enables the simultaneous peeling of three nodes in the $(i+1)$-th iteration. Without this pruning strategy, an additional iteration would be essential to peel vertex u, underscoring our method's enhanced efficiency in the peeling process.

5.2 Local Peeling Optimization

It is important to note that each peeling iteration often results in disparate structures. This is primarily caused by the peeling of certain vertices, which leaves their neighboring structures disconnected and fragmented. This not only affects the density of the detected subgraph but also lowers the peeling threshold. Lemma 5.2 establishes a property of vertices within a dense graph.

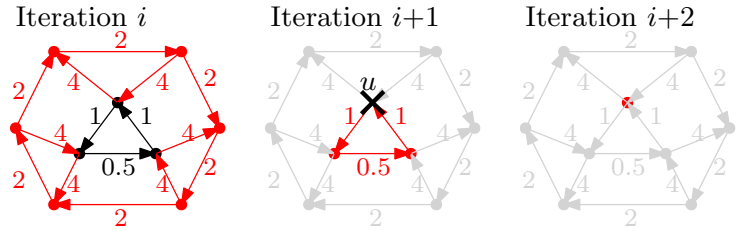


Fig. 7. Demonstrating Long-Tail Pruning. Nodes highlighted in red are targeted for removal in the current iteration, while the nodes shaded in gray indicate those that have been peeled.

Algorithm 4: DupinLPO: Local Peeling Optimization

Input: $G = (V, E)$, a density metric $g(S)$, and $\epsilon \geq 0$
Output: A subset of vertices S^p representing the densest subgraph

$S_0 \leftarrow V$ /* Initialize the subset with all vertices */
$\tau_{\max} \leftarrow 0$ /* Initialize the global threshold */
$i \leftarrow 1$ /* Initialize the index */
while $S_{i-1} \neq \emptyset$ **do**
  parallel_for $u_j \in S_{i-1}$ **do**
    $a_j \leftarrow \text{vsusp}(u_j)$
  parallel_for $(u_j, u_k) \in E$ ***and*** $u_j, u_k \in S_{i-1}$ **do**
    $c_{jk} \leftarrow \text{esusp}(u_j, u_k)$
  $f(S_{i-1}) \leftarrow \texttt{parallel\_sum}(a_j) + \texttt{parallel\_sum}(c_{jk})$
  $g(S_{i-1}) \leftarrow f(S_{i-1}) / |S_{i-1}|$
  $\tau_{\max} \leftarrow \max\{\tau_{\max}, \frac{g(S_{i-1})}{k(1+\epsilon)}\}$ /* Refine the global threshold */
  $\tau_1 \leftarrow \max\{\tau_{\max}, k(1+\epsilon) \cdot g(S_{i-1})\}$ /* Peeling threshold */
  $U_1 \leftarrow \emptyset$ /* Initialize the first removal set */
  parallel_for $u \in S_{i-1}$ **do**
    if $w_u(S_{i-1}) \leq \tau_1$ **then**
      $U_1 \leftarrow U_1 \cup \{u\}$ /* Vertices to be peeled */
  $S_i \leftarrow S_{i-1} \setminus U_1$ /* Update the subset */
  while $\exists u \in S_i$ *such that* $w_u(S_i) < g(S_i)$ **do**
    $\tau_2 \leftarrow \max\{\tau_{\max}, g(S_i)\}$ /* Refine the local threshold */
    $U_2 \leftarrow \emptyset$ /* Initialize the second removal set */
    parallel_for $u \in S_i$ **do**
      if $w_u(S_i) < \tau_2$ **then**
        $U_2 \leftarrow U_2 \cup \{u\}$ /* Vertices to be peeled */
    $S_i \leftarrow S_i \setminus U_2$ /* Trim the subset */
    $i \leftarrow i + 1$ /* Increment the index */
return $\arg\max_{S_i} g(S_i)$ /* The subset that maximizes $g(S)$ */

Lemma 5.2. *$\forall u \in S$, if $w_u(S) < g(S)$, $g(S \setminus \{u\}) > g(S)$.*

Proof. Suppose u is peeled from S, a denser graph is obtained:

$$g(S \setminus \{u\}) = \frac{f(S) - w_u(S)}{|S|-1} > \frac{f(S) - g(S_i)}{|S|-1} = \frac{f(S) - g(S)}{|S^*|-1} = \frac{f(S) - \frac{f(S)}{|S|}}{|S|-1} = g(S) \tag{14}$$

□

Table 4. Statistics of real-world datasets.

Datasets	Name	|V|	|E|	avg. degree	Type
GFG	gfg	3,646,185	28,635,763	17	Transaction
soc-twitter	soc	28,504,110	531,000,244	18	Social network
Web-uk-2005	uk	39,454,748	936,364,284	24	Web graph
Twitter-rv	rv	41,652,230	1,468,365,182	35	Social network
kron-logn21	kron	1,544,088	91,042,012	58	Cheminformatics
Web-sk-2005	sk	50,636,151	1,949,412,601	38	Web graph
links-anon	la	52,579,682	1,963,263,821	37	Social network
biomine	bio	1,508,587	32,761,889	22	Biologic graph

According to Lemma 5.2, we introduce local peeling optimization within the iteration. After each peeling iteration, if the peeling weight of a vertex is smaller than the current density, Dupin will trim it. We implement a local peeling optimization (Lines 18-24, Algorithm 4) before the next iteration. After a vertex u is peeled, Dupin invokes updateNgh to update the peeling weights of u's neighboring nodes. If any of these neighbors have a peeling weight less than the current density, they are also peeled, thereby increasing the density of the current iteration. Consequently, when an iteration yields globally optimal results, a denser structure is obtained. The benefits of this approach extend beyond density improvements; performance is also enhanced. By trimming these vertices, we reduce the number of vertices to traverse in subsequent iterations, and a higher density enables more efficient iterations due to the possibility of using larger peeling thresholds.

Lemma 5.3. *If $u_i \in S^p$ is the first vertex in S^* removed by* Dupin, *u_i will not be removed during the local peeling optimization process.*

Proof. Suppose u_i is trimmed during the local peeling optimization process. Let S' be the set of vertices left before u_i is trimmed. By definition, we have:

$$w_{u_i}(S') < g(S^*).$$

Since u_i is the first vertex in S^* to be removed, $S^* \subseteq S'$, therefore:

$$w_{u_i}(S^*) \leq w_{u_i}(S') < g(S^*).$$

This contradicts Lemma 5.2, which states that $\forall u_i \in S^*$:

$$w_{u_i}(S^*) \geq g(S^*).$$

Otherwise, removing u_i from S^* would yield a better solution, contradicting the assumption that S^* is optimal. Thus, u_i cannot be trimmed. □

Due to Lemma 5.3, u_i will only be removed during the peeling process. Hence, the approximation ratios (Theorem 4.2) for different density metrics are preserved.

6 Experimental Study

6.1 Experimental Setup

Our experiments are run on a machine with the following specifications: Intel Xeon X5650 CPU[2], 512 GB RAM, and running on Ubuntu 20.04 LTS. The implementation is memory-resident and developed in C++. All codes are compiled with optimization flag -*O*3.

Datasets. We report the datasets used in our experiment in Table 4. One industrial dataset is from Grab (gfg), which represents a real-world scenario of fraud detection in a large-scale transaction

[2]The Xeon X5650 was selected to ensure a fair comparison with the setup reported in Spade's original work. To assess the impact of different hardware, we evaluated the performance of Spade, Dupin, FWA, GBBS, and PBBS on two CPUs in Appendix A of [28], showing that Dupin achieves better speedup compared to the other systems in the modern hardware.

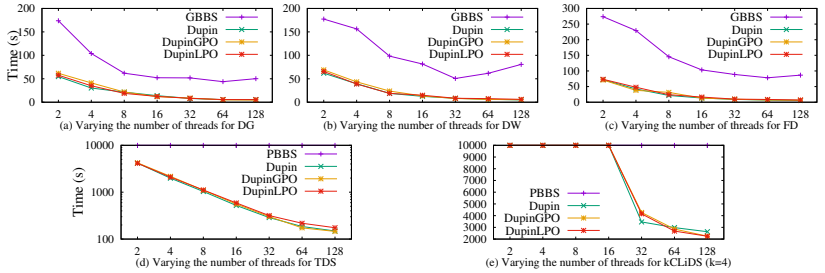


Fig. 8. Scalability of GBBS, PBBS, and Dupin by varying the number of the threads.

network. This dataset is particularly relevant as it contains diverse transaction patterns and user behaviors, reflecting the complexities of real-world fraud detection. Datasets `links-anon(la)` and `Twitter-rv (rv)` were obtained from Stanford Network Dataset Collection [33], providing insights into social network structures. Datasets `kron-logn21(kron)`, `soc-twitter(soc)`, `Web-uk-2005(uk)`, and `Web-sk-2005(sk)` were obtained from Network Data Repository [42], representing various web graph characteristics. Dataset `bio` was from the BIOMINE [39], which offers biological network structures. The selection of these datasets was based on their ability to represent different types of graphs, thus enhancing the generalizability of our results.

Baseline Methods. We evaluate five commonly used density metrics pivotal in network structure analysis: DG [6], DW [23], FD [24], TDS [50], and kCLiDS [14]. These metrics are implemented within our framework, Dupin, and their performance is compared against several existing frameworks: Spade [27], kCLIST [14], GBBS [16], PBBS [45], PKMC [35], FWA [15], and ALENEX [48]. Specifically, kCLIST and PBBS are parallel algorithms for kCLiDS. Since GBBS does not natively support weighted graphs, we precompute the peeling weights offline and import them into GBBS to enable support for DW and FD; the reported runtime excludes this preprocessing time. Additionally, FWA, ALENEX, and PKMC support the DG metric, and we have extended their algorithms to support DW and FD by modifying their codebases accordingly. For TDS and kCLiDS, we adapted the API of Spade for implementation. We utilized the latest codebase Spade+ of Spade to ensure consistency and reliability in our performance measurements. To standardize the comparison, we denote the incremental peeling process—which iteratively removes graph elements based on density scores—as Spade-X, where X corresponds to each metric. This follows the experimental conditions in [27], with the batch size set to 1K as per their default setting; we report the average runtime per batch for Spade. Our parallel peeling algorithms, incorporating global and local peeling optimizations, are denoted as DupinGPO-X and DupinLPO-X, respectively.

Default Settings. In our experiments, the parameter ϵ is set to 0.1 by default, and the number of threads t is configured to 128. These settings are used consistently unless otherwise specified.

6.2 Efficiency of Dupin

Spade **vs.** Dupin. Our initial comparison focuses on the performance disparity between the incremental and parallel peeling algorithms. Specifically, Dupin-DG (resp. Dupin-DW, Dupin-FD, Dupin-TDS, and Dupin-kCLiDS) demonstrates a speedup factor of 6.97 (resp. 7.71, 7.71, 7.65, and 8.46) over Spade-DG (resp. Spade-DW, Spade-FD, Spade-TDS, and Spade-kCLiDS) as shown in Table 5 and Table 6. Notably, Spade-TDS and Spade-kCLiDS only complete computations within 7,200 seconds on `gfg`. The bottleneck for Spade arises in larger graphs where initial calculations of the triangle and k-Clique counting are unable to finish. Dupin outpaces Spade because Spade suffers from the frequent reordering of peeling sequences when there are many increments, which proves to be slower than recalculating.

GBBS **vs.** Dupin**.** Among the parallel methods for DG, DW, and FD, FWA, ALENEX, and PKMC are several to dozens of times slower than GBBS. Therefore, we select GBBS as a representative for comparison with Dupin. The runtimes of the other three methods are provided in Table 5. On average, Dupin-DG, Dupin-DW, and Dupin-FD demonstrated speedup factors of 6.33, 12.51, and 17.07 times, respectively, over GBBS-DG, GBBS-DW, and GBBS-FD. This performance advantage is attributed to the fundamental difference in the peeling process between the two frameworks. In GBBS, the basic unit of peeling is a bucket, where nodes in the same bucket have an identical peeling weight. This design limits parallelism in weighted graphs, such as those using DW and FD, where many buckets contain only a single node. In contrast, Dupin peels batches of nodes at each step and maintains a global peeling threshold to enhance the efficiency of each iteration and eliminate long-tail iterations. This approach achieves higher parallelism and consistently outperforms GBBS, particularly in weighted graph scenarios where the speedup is even more pronounced.

kCLIST **vs.** PBBS **vs.** Dupin**.** Both kCLIST and PBBS support the density metrics TDS and kCLiDS. In our comparisons, Dupin demonstrates notable performance advantages. Specifically, Dupin-TDS is faster than kCLIST-TDS and PBBS-TDS by 39.85 and 62.71 times, respectively. Furthermore, Dupin-kCLiDS also outperforms kCLIST-kCLiDS by 4.16 times. A significant finding is that PBBS-kCLiDS fails to complete computations within 7200 seconds on most datasets, highlighting the efficiency of Dupin in handling more complex metrics under stringent time constraints.

Table 5. Overall efficiency evaluation conducted using 128 threads for density metrics DG, DW, and FD. The notation 'TLE' signifies "Time Limit Exceeded," indicating that the task did not finish within 7,200 seconds.

Methods	**Dataset**	DG	DW	FD	**Dataset**	DG	DW	FD
Spade	soc	23.46	28.10	30.67	sk	182.28	215.27	210.26
GBBS		10.01	23.28	35.43		13.95	35.91	48.80
PKMC		103.16	105.77	106.96		388.85	393.01	431.04
FWA		704	1,241	1,381		3,092	4,934	4,716
ALENEX		138.73	120.19	128.81		411.75	488.49	473.45
Dupin		**1.79**	**2.26**	**2.38**		**3.87**	**3.93**	**4.25**
Spade	uk	67.08	93.66	83.20	la	175.07	242.26	224.47
GBBS		7.26	27.15	50.29		50.21	80.66	50.29
PKMC		167.76	168.81	187.61		946.64	1048.48	1024.23
FWA		1,794	TLE	TLE		4995	TLE	TLE
ALENEX		238.36	209.30	213.80		215.98	154.17	194.06
Dupin		**3.38**	**3.57**	**3.56**		**4.55**	**4.41**	**5.16**
Spade	rv	135.89	135.05	132.61	bio	1.99	2.02	2.11
GBBS		32.51	62.29	71.67		2.55	5.04	5.21
PKMC		659.04	659.15	693.83		11.38	11.39	12.97
FWA		3569	TLE	TLE		62	393	428
ALENEX		158.64	132.88	158.79		62.16	68.22	63.62
Dupin		**3.34**	**3.76**	**3.89**		**0.15**	**0.20**	**0.23**
Spade	gfg	2.30	2.62	2.70	kron	5.08	5.36	5.61
GBBS		0.41	2.10	5.74		1.47	6.62	9.58
PKMC		15.96	15.00	16.25		34.47	35.71	37.23
FWA		133	186	173		254	344	372
ALENEX		4.13	4.01	4.75		85.8	83.05	82.98
Dupin		**0.29**	**0.33**	**0.35**		**0.18**	**0.36**	**0.23**

Impact of DupinGPO**.** Next, we evaluate the acceleration effect of DupinGPO, using Dupin as a baseline for comparison. Notably, DupinGPO-DG is found to be 1.14 times faster, DupinGPO-DW is 1.11 times faster, DupinGPO-TDS is 1.10 times faster, and DupinGPO-kCLiDS is 1.35 times

Table 6. Overall efficiency evaluation conducted using 128 threads for density metrics TDS and kCLiDS.

Methods	**Dataset**	TDS	kCLiDS	**Dataset**	TDS	kCLiDS
Spade	soc	TLE	TLE	sk	TLE	TLE
kCLIST		1516	1444		TLE	TLE
PBBS		3524.11	TLE		TLE	TLE
Dupin		**32.59**	**283.70**		**42.73**	**2636.17**
Spade	uk	TLE	TLE	la	TLE	TLE
kCLIST		494	447		10663	8003
PBBS		2900.83	TLE		TLE	TLE
Dupin		**16.66**	**186.31**		**145.72**	**2241.69**
Spade	rv	TLE	TLE	bio	TLE	TLE
kCLIST		5561	4655		230.0	255.0
PBBS		TLE	TLE		225.06	TLE
Dupin		**96.32**	**1009.45**		**6.25**	**82.54**
Spade	gfg	5.66	5.84	kron	TLE	TLE
kCLIST		10	9		406	450
PBBS		1.37	0.79		330.18	TLE
Dupin		**0.74**	**0.69**		**11.69**	**138.63**

Table 7. Overall density evaluation for DG, DW, and FD.

Methods	**Dataset**	DG	DW	FD	**Dataset**	DG	DW	FD
Spade	soc	1,307	63,372	16,947	sk	2,257	109,097	33,741
GBBS		1,307	63,372	16,883		2,257	109,097	28,118
PKMC		1,053	54,373	15,363		1,954	95,241	25,868
FWA		1,307	63,372	16,883		2,257	109,097	28,188
ALENEX		1,307	63,372	16,883		2,257	109,097	28,188
Dupin		1,286	59,379	16,234		2,235	98,669	27,067
Spade	uk	426	27,812	10,155	la	1,877	89,641	22,198
GBBS		426	27,812	8,987		1,877	89,641	21,774
PKMC		375	24,279	8,268		1,512	83,049	19,637
FWA		486	TLE	TLE		1,877	TLE	TLE
ALENEX		486	27,812	8,987		1,877	89,641	21,774
Dupin		309	24,796	8,424		1,843	87,671	20,610
Spade	rv	1,643	74,779	22,678	bio	777	36,446	13,039
GBBS		1,643	74,779	20,749		777	36,446	12,669
PKMC		1,437	67,328	16,753		721	32,284	11,283
FWA		1,643	TLE	TLE		787	36,446	12,669
ALENEX		1,643	74,779	20,749		787	36,446	12,669
Dupin		1,518	71,058	18,115		699	31,172	10,404
Spade	gfg	28	1,432	5,369	kron	1,177	53,539	15,381
GBBS		28	1,432	5,018		1,177	53,539	14,861
PKMC		28	1,396	4,782		1,169	49,983	12,829
FWA		28	1,432	5,108		1,177	53,539	14,861
ALENEX		28	1,432	5,108		1,177	53,539	14,861
Dupin		26	1,405	4,879		1,177	52,695	13,912

faster than their respective Dupin implementations. The acceleration is particularly significant for kCLiDS on larger datasets, such as soc, rv, kron, and sk. DupinGPO-kCLiDS is 1.40, 1.54, 1.51, and 1.61 times faster than Dupin-kCLiDS on soc, rv, kron, and sk, respectively. This is attributed

Table 8. Overall density evaluation for TDS and kCLiDS.

Methods	**Dataset**	TDS	kCLiDS	**Dataset**	TDS	kCLiDS
Spade	soc	TLE	TLE	sk	TLE	TLE
kCLIST		1,525,517	645,536,400		TLE	TLE
PBBS		1,994,617	TLE		TLE	TLE
Dupin		1,533,939	606,708,980		9,995,529	14,890,921,876
Spade	uk	TLE	TLE	la	TLE	TLE
kCLIST		188,524	44,272,600		3,865,986	3,306,718,000
PBBS		304,577	TLE		TLE	TLE
Dupin		187,260	33,823,488		3,974,028	3,167,144,896
Spade	rv	TLE	TLE	bio	TLE	TLE
kCLIST		3,561,222	3,280,083,600		1,043,100	542,860,800
PBBS		TLE	TLE		1,187,388	TLE
Dupin		3,769,671	2,997,257,620		1,138,083	544,047,728
Spade	gfg	0	0	kron	TLE	TLE
kCLIST		0	0		1,447,859	424,908,400
PBBS		0	0		1,447,859	TLE
Dupin		0	0		1,447,788	426,677,504

to the greater number of iterations required due to the larger dataset sizes and the higher count of k-Cliques, necessitating more peeling iterations. In such scenarios, DupinGPO effectively prunes the tail iterations early, thereby reducing the number of rounds needed.

Impact of DupinLPO**.** We next compared the efficiency of Dupin and DupinLPO. On average, though DupinLPO-DG was 10.1% slower than Dupin-DG, the speed is still comparable. For example, on the soc dataset, DupinLPO-DG was 5.9% faster than Dupin-DG. This is because DupinLPO requires some lock operations to update the peeling weights of neighboring vertices during local peeling optimization. However, the advantage is that DupinLPO can detect up to 26.3% denser subgraphs, as detailed in Section 6.3. On larger datasets, such as soc, rv, and la, the vertices being trimmed have relatively low correlations with each other, so the impact of locks is minimal. For example, on the soc (resp. rv), DupinLPO-DG was 5.9% (resp. 9.9%) faster than Dupin-DG. In contrast, on smaller datasets, such as bio and kron, the extra cost of locks is higher.

Impact of the Number of Threads t**.** All methods are influenced by the concurrency level, *i.e.,* the number of threads. Generally, the greater the number of threads, the faster the execution speed. In our experiments, using the la dataset as a benchmark, we varied the number of threads t from 2 to 128 to observe the impact on runtime performance. For GBBS-DG, runtime stabilizes and accelerates by 3.33 times as thread count increases from 2 to 8. For GBBS-DW, performance deteriorates when the thread count exceeds 32, but from 2 to 32 threads, it accelerates by 3.50 times. Similarly, GBBS-FD shows no significant change in runtime beyond 32 threads, with an acceleration of 3.09 times from 2 to 32 threads. GBBS-TDS and GBBS-kCLiDS fail to complete computations on the la dataset. In contrast, all five methods tested on Dupin exhibit good scalability. As the number of threads increases from 2 to 128, Dupin-DG accelerates by 9.91 times, Dupin-DW by 12.69 times, Dupin-FD by 13.03 times, and Dupin-TDS by 28.06 times. Although Dupin-kCLiDS fails to produce results when the thread count is below 16, it accelerates by 1.32 times from 32 to 128 threads. These results demonstrate Dupin's strong scalability across varied threading levels.

CPU Utilization (Figure 9). Using DG as an example, as the number of threads increases from 2 to 64, the stall rates of GBBS grow more rapidly than those of Dupin, demonstrating Dupin's superior scalability. In particular, at 64 threads, the busy rate of Dupin is approximately 2.49 times

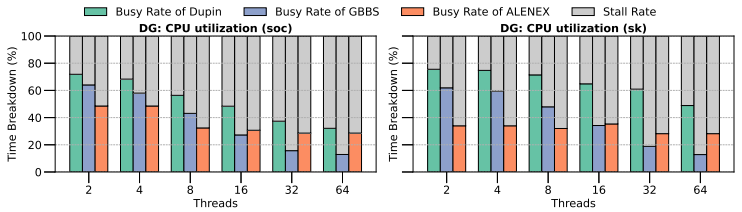


Fig. 9. CPU Utilization *w.r.t.* DG.

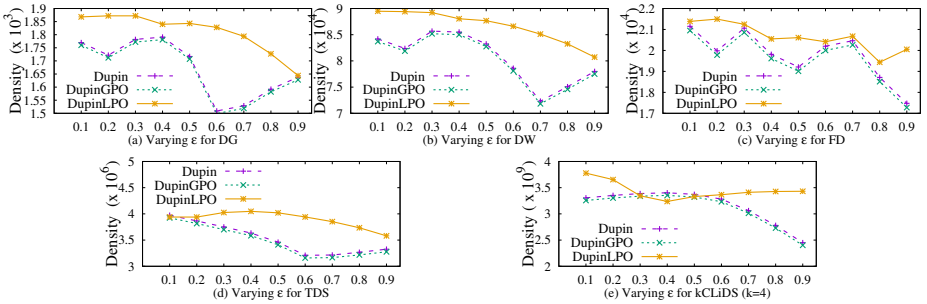


Fig. 10. Effectiveness of Dupin by varying ϵ.

(resp. 3.81 times) that of GBBS and 1.04 times (resp. 1.67 times) that of ALENEX on soc (resp. sk). Additional CPU utilization data across datasets are provided in Appendix A of [28].

6.3 Effectiveness of Dupin

In addition to assessing performance speedup, we evaluate the density of the subgraphs detected by different frameworks. For parallel methods, we consider GBBS, PBBS, and kCLIST, because they generally detect subgraphs with higher density than those detected by other methods across most datasets. For incremental methods, we use Spade as a representative. Detailed density comparisons are presented in Table 7 and Table 8.

GBBS **vs.** Spade **vs.** Dupin**.** Firstly, we compare the densities of the subgraphs identified by GBBS-DG (respectively GBBS-DW, and GBBS-FD) to those detected by Dupin in parallel execution. Our experimental results indicate that Dupin maintains comparable subgraph densities with GBBS, signifying that the increase in computational efficiency does not compromise the accuracy of the detected subgraphs. Specifically, the density of the subgraph detected by GBBS-DG (respectively GBBS-DW, and GBBS-FD) is 7.1% (resp. 6.5% and 7.4%) greater than that detected by Dupin-DG (respectively Dupin-DW, and Dupin-FD) on average. Spade shares similar densities with GBBS. Due to Spade's tendency to accumulate errors over time, its density can become inflated. We will explain its limitations in practical applications in more detail in the case study.

kCLIST **vs.** PBBS **vs.** Dupin**.** a) For kCLiDS, PBBS-kCLiDS could not complete on most of our test datasets, so we did not compare their densities. kCLIST-kCLiDS could not complete on the sk dataset. On the other datasets, Dupin-kCLiDS's density was only 7.0% smaller than kCLIST-kCLiDS. On some datasets, such as bio and kron, Dupin-kCLiDS even detected subgraphs with greater density. b) For TDS, PBBS-TDS could detect subgraphs on smaller graphs but failed on billion-scale graphs such as rv, sk, and la. Dupin-TDS's density was 2.0% greater than kCLIST-TDS. Therefore, we conclude that Dupin achieves similar subgraph density as existing parallel methods.

Ablation Study of Dupin**.** This experiment delves into the effectiveness of our optimizations, DupinGPO and DupinLPO. Our analysis primarily focuses on comparing the densities of dense subgraphs detected by both DupinGPO and DupinLPO in various settings. DupinGPO achieves

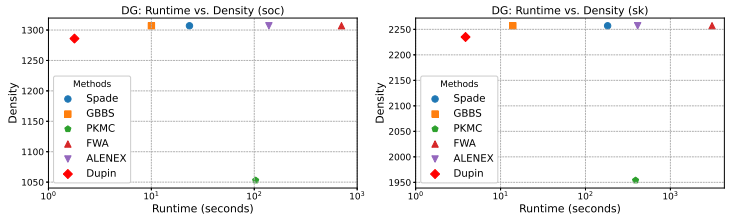


Fig. 11. Comparison of runtime and density across different methods on two datasets, *w.r.t.* DG.

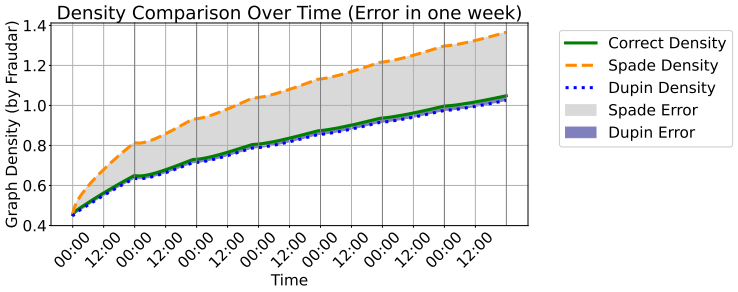


Fig. 12. Weekly Graph Density Comparison in Grab— Shows Correct Density (green), Spade Density (orange dashed), and Dupin Density (blue dotted) over one week. Gray and navy shaded areas indicate error margins for Spade and Dupin, highlighting their deviations from the ideal density.

the same density as Dupin across five density metrics because it trims the long tail, enhancing efficiency. For DupinLPO-DG (resp. DupinLPO-DW, DupinLPO-FD, DupinLPO-TDS, and DupinGPO-kCLiDS), we observe that the detected subgraphs exhibit densities 11.1% (resp. 7.3%, 8.0%, 1.0%, and 26.3%) greater than those detected by Dupin-DG, Dupin-DW, Dupin-FD, Dupin-TDS, and Dupin-kCLiDS, respectively, demonstrating notable outcomes. This is because DupinLPO trims the graph in each iteration, resulting in denser subgraphs compared to Dupin.

Impact of the Approximation Ratio ϵ. The approximation ratio ϵ serves to control accuracy. In our experiment, we vary ϵ from 0.1 to 1 to investigate its influence on the accuracy of Dupin. We observe that as ϵ increases from 0.1 to 1, the density of subgraphs detected by Dupin, DupinGPO, and DupinLPO generally decreases. For DG, the densities of Dupin-DG, DupinGPO-DG, and DupinLPO-DG decrease by 22.7%, 22.7%, and 12.0%, respectively. The trends for DW and DG are similar, with the worst performance observed at $\epsilon = 0.6$ for both density metrics. The trends for FD, TDS, and kCLiDS are also similar. DupinLPO detects denser subgraphs across most density metrics and is less affected by ϵ, due to its iterative trimming. For example, for FD, DupinLPO's density decreases by only 6.2%, while DupinGPO and Dupin's densities decrease by 17.4%.

Runtime-Density Trade-off Analysis. Figure 11 shows runtime versus the density for six methods Spade, GBBS, PKMC, FWA, ALENEX, and Dupin over two representative datasets (soc and sk). Spade and FWA achieve high densities at the cost of longer runtimes, PKMC is fast but yields lower densities, while GBBS and ALENEX strike a moderate balance. Notably, Dupin consistently finishes the fastest with competitive densities, making it ideal for real-time or large-scale applications. Additional results for other datasets and density measures are provided in Appendix A of [28].

6.4 Case Study

The comparison in real-world scenarios focusing on accuracy, latency, and prevention ratio.

Accuracy. As discussed in Section 1, although Spade can quickly respond to new transactions, it tends to accumulate errors over time. Our analysis utilizes transaction data from Grab, with the

	DG		DW		FD		TDS	
	$\mathcal{L}$	$\mathcal{R}$	$\mathcal{L}$	$\mathcal{R}$	$\mathcal{L}$	$\mathcal{R}$	$\mathcal{L}$	$\mathcal{R}$
Dupin	3.10	78.8%	3.54	86.1%	3.59	94.5%	2145.00	32%
Spade	165.20	58%	235.63	63%	197.61	45.3%	TLE	TLE
GBBS	927.88	12%	TLE	TLE	6014.00	3.0%	-	-

Table 9. Latency vs. Prevention Ratio by Method: A method's performance in latency ($\mathcal{L}$: seconds) against its prevention ratio ($\mathcal{R}$), detailing the effectiveness of various computational strategies in real-time environments.

transaction network comprising $|E|=$ 2 billion edges. Under the incremental computation framework of Spade, the assumption is made that the weights of existing edges remain constant, thereby neglecting the impact of newly inserted edges on these weights, leading to error accumulation. For instance, using FD density metric, we observe that errors accumulate up to 24.4% within a day and increase to 30.3% over a week, as illustrated in Figure 12. Although Dupin sacrifices some accuracy to achieve parallel computational efficiency, our experiments show that Dupin's errors remain relatively stable between 3.0% and 5.0%. Hence, the detection precision of Spade declines from 87.9% to 68.3%, while Dupin's precision remains above 86.0%.

Latency and Prevention Ratio. If a transaction is identified as fraudulent, subsequent related transactions are blocked to mitigate potential losses. We define the ratio of successfully prevented suspicious transactions to the total number of suspicious transactions as $\mathcal{R}$. As shown in Table 9, the prevention ratio $\mathcal{R}$ continues to decrease as latency increases on Grab's datasets. Using the default density metric, FD, in Grab, our results show that Dupin-FD prevents 94.5% of fraudulent activities, GBBS-FD prevents 3.0%, and Spade-FD prevents 45.3%. The reason for the lower performance of GBBS-FD is its low parallelism in the peeling process, which results in long peeling times and thus an average latency of 6,014 seconds, delaying the prevention of many fraudulent activities. Although Spade-FD performs well in reordering techniques on smaller datasets, in industrial scenarios with billion-scale graphs, latency significantly worsens due to the extensive reordering range required in the peeling sequences as normal users transition to fraudsters, increasing the cost of incremental computations. Additional tests were conducted with other density metrics; for DG, Dupin-DG prevents 78.8%, while other methods prevent varying amounts. Similarly, for DW, Dupin-DW prevents 86.1%, while other methods demonstrate different prevention capabilities.

7 Conclusion and Future Work

In this paper, we presented Dupin, a novel parallel fraud detection framework tailored for massive graphs. Our experiments demonstrated that Dupin's advanced peeling algorithms and long-tail pruning techniques, DupinGPO and DupinLPO, significantly enhance the detection efficiency of dense subgraphs without sacrificing accuracy. Comparative studies with GBBS, PBBS, kCLIST, Spade, and Dupin highlighted Dupin's superior performance in terms of computational efficiency and accuracy. Through case studies, we validated that Dupin achieves lower latency and higher detection accuracy compared to existing fraud detection systems on billion-scale graphs.

Acknowledgement. This research is supported by the National Research Foundation, Singapore under its AI Singapore Programme (AISG Award No: AISG2-TC-2021-002), the Ministry of Education, Singapore, under its Academic Research Fund Tier 2 (Award No: MOE2019-T2-2-065), and the Singapore Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 1 grant, 23-SIS-SMU-063. Yuchen Li is partially supported by the Lee Kong Chian fellowship.

References

[1] Distil networks: The 2019 bad bot report. https://www.bluecubesecurity.com/wp-content/uploads/bad-bot-report-2019LR.pdf.

[2] B. Bahmani, R. Kumar, and S. Vassilvitskii. Densest subgraph in streaming and mapreduce. *Proceedings of the VLDB Endowment*, 5(5), 2012.

[3] Y. Ban, X. Liu, T. Zhang, L. Huang, Y. Duan, X. Liu, and W. Xu. Badlink: Combining graph and information-theoretical features for online fraud group detection. *arXiv preprint arXiv:1805.10053*, 2018.

[4] A. Beutel, W. Xu, V. Guruswami, C. Palow, and C. Faloutsos. Copycatch: stopping group attacks by spotting lockstep behavior in social networks. In *Proceedings of the 22nd international Conf. on World Wide Web*, pages 119–130, 2013.

[5] D. Boob, Y. Gao, R. Peng, S. Sawlani, C. Tsourakakis, D. Wang, and J. Wang. Flowless: Extracting densest subgraphs without flow computations. In *Proceedings of The Web Conf. 2020*, pages 573–583, 2020.

[6] M. Charikar. Greedy approximation algorithms for finding dense components in a graph. In *International Workshop on Approximation Algorithms for Combinatorial Optimization*, pages 84–95. Springer, 2000.

[7] 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*, volume 1913 of *Lecture Notes in Computer Science*, pages 84–95. Springer, 2000.

[8] C. Chekuri, K. Quanrud, and M. R. Torres. Densest subgraph: Supermodularity, iterative peeling, and flow. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1531–1555. SIAM, 2022.

[9] J. Chen and Y. Saad. Dense subgraph extraction with application to community detection. *IEEE Transactions on knowledge and data engineering*, 24(7):1216–1230, 2010.

[10] Y. Chen, J. Jiang, S. Sun, B. He, and M. Chen. Rush: Real-time burst subgraph detection in dynamic graphs. *Proceedings of the VLDB Endowment*, 17(11):3657–3665, 2024.

[11] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on computing*, 14(1):210–223, 1985.

[12] L. Chu, Y. Zhang, Y. Yang, L. Wang, and J. Pei. Online density bursting subgraph detection from temporal graphs. *Proceedings of the VLDB Endowment*, 12(13):2353–2365, 2019.

[13] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[14] M. Danisch, O. Balalau, and M. Sozio. Listing k-cliques in sparse real-world graphs. In *Proceedings of the 2018 World Wide Web Conf.*, pages 589–598, 2018.

[15] M. Danisch, T.-H. H. Chan, and M. Sozio. Large scale density-friendly graph decomposition via convex programming. In *Proceedings of the 26th International Conf. on World Wide Web*, pages 233–242, 2017.

[16] L. Dhulipala, J. Shi, T. Tseng, G. E. Blelloch, and J. Shun. The graph based benchmark suite (gbbs). In *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, pages 1–8, 2020.

[17] Y. Dourisboure, F. Geraci, and M. Pellegrini. Extraction and classification of dense communities in the web. In *Proceedings of the 16th international Conf. on World Wide Web*, pages 461–470, 2007.

[18] A. Epasto, S. Lattanzi, and M. Sozio. Efficient densest subgraph computation in evolving graphs. In *Proceedings of the 24th international Conf. on world wide web*, pages 300–310, 2015.

[19] Y. Fang, R. Cheng, S. Luo, and J. Hu. Effective community search for large attributed graphs. *Proceedings of the VLDB Endowment*, pages 1233–1244, 2016.

[20] D. Gibson, R. Kumar, and A. Tomkins. Discovering large dense subgraphs in massive graphs. In *Proceedings of the 31st international Conf. on Very large data bases*, pages 721–732. Citeseer, 2005.

[21] A. Gionis and C. E. Tsourakakis. Dense subgraph discovery: Kdd 2015 tutorial. In *Proceedings of the 21th ACM SIGKDD International Conf. on Knowledge Discovery and Data Mining*, pages 2313–2314, 2015.

[22] A. V. Goldberg. Finding a maximum density subgraph. Technical report, University of California, Berkeley, USA, 1984. EECS Technical Memorandum Series.

[23] N. V. Gudapati, E. Malaguti, and M. Monaci. In search of dense subgraphs: How good is greedy peeling? *Networks*, 77(4):572–586, 2021.

[24] B. Hooi, H. A. Song, A. Beutel, N. Shah, K. Shin, and C. Faloutsos. Fraudar: Bounding graph fraud in the face of camouflage. In *Proceedings of the 22nd ACM SIGKDD international Conf. on knowledge discovery and data mining*, pages 895–904, 2016.

[25] M. Jaggi. Revisiting frank-wolfe: Projection-free sparse convex optimization. In *International Conf. on machine learning*, pages 427–435. PMLR, 2013.

[26] J. Jiang, Y. Chen, B. He, M. Chen, and J. Chen. Spade+: A generic real-time fraud detection framework on dynamic graphs. *IEEE Transactions on Knowledge and Data Engineering*, 2024.

[27] J. Jiang, Y. Li, B. He, B. Hooi, J. Chen, and J. K. Z. Kang. Spade: A real-time fraud detection framework on evolving graphs. *Proceedings of the VLDB Endowment*, 16(3):461–469, 2022.
[28] J. Jiang, S. Yao, Y. Li, Q. Wang, B. He, and M. Chen. Dupin: A parallel framework for densest subgraph discovery in fraud detection on massive graphs (technical report). *arXiv preprint arXiv:2504.09311*, 2025.
[29] M. Jiang, P. Cui, A. Beutel, C. Faloutsos, and S. Yang. Inferring strange behavior from connectivity pattern in social networks. In *Pacific-Asia Conf. on knowledge discovery and data mining*, pages 126–138. Springer, 2014.
[30] B. Klimat. The enron corpus: A new datasct tor email classification rcscarch. In *Proceedings of the European Conf. on Machinc Learning (ECML). 2004*, 2004.
[31] S. Kumar, W. L. Hamilton, J. Leskovec, and D. Jurafsky. Community interaction and conflict on the web. In *Proceedings of the 2018 world wide web Conf.*, pages 933–943, 2018.
[32] V. E. Lee, N. Ruan, R. Jin, and C. Aggarwal. A survey of algorithms for dense subgraph discovery. In *Managing and mining graph data*, pages 303–336. Springer, 2010.
[33] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.
[34] X. Liao, Q. Liu, J. Jiang, X. Huang, J. Xu, and B. Choi. Distributed d-core decomposition over large directed graphs. *Proc. VLDB Endow.*, 15(8):1546–1558, apr 2022.
[35] W. Luo, Z. Tang, Y. Fang, C. Ma, and X. Zhou. Scalable algorithms for densest subgraph discovery. In *2023 IEEE 39th International Conf. on Data Engineering (ICDE)*, pages 287–300. IEEE, 2023.
[36] C. Ma, Y. Fang, R. Cheng, L. V. Lakshmanan, and X. Han. A convex-programming approach for efficient directed densest subgraph discovery. In *Proceedings of the 2022 International Conf. on Management of Data*, pages 845–859, 2022.
[37] M. Magdon-Ismail, M. Goldberg, W. Wallace, and D. Siebecker. Locating hidden groups in communication networks using hidden markov models. In *Intelligence and Security Informatics: First NSF/NIJ Symposium, ISI 2003, Tucson, AZ, USA, June 2–3, 2003 Proceedings 1*, pages 126–137. Springer, 2003.
[38] M. Mitzenmacher, J. Pachocki, R. Peng, C. Tsourakakis, and S. C. Xu. Scalable large near-clique detection in large-scale networks via sampling. In *Proceedings of the 21th ACM SIGKDD International Conf. on Knowledge Discovery and Data Mining*, pages 815–824, 2015.
[39] V. Podpečan, Ž. Ramšak, K. Gruden, H. Toivonen, and N. Lavrač. Interactive exploration of heterogeneous biological networks with biomine explorer. *Bioinformatics*, 06 2019.
[40] H. Qin, R.-H. Li, Y. Yuan, G. Wang, L. Qin, and Z. Zhang. Mining bursting core in large temporal graphs. *Proceedings of the VLDB Endowment*, 2022.
[41] Y. Ren, H. Zhu, J. Zhang, P. Dai, and L. Bo. Ensemfdet: An ensemble approach to fraud detection based on bipartite graph. In *2021 IEEE 37th International Conf. on Data Engineering (ICDE)*, pages 2039–2044. IEEE, 2021.
[42] R. A. Rossi and N. K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.
[43] A. E. Sarıyüce and A. Pinar. Peeling bipartite networks for dense subgraph discovery. In *Proceedings of the Eleventh ACM International Conf. on Web Search and Data Mining*, pages 504–512, 2018.
[44] S. B. Seidman. Network structure and minimum degree. *Social networks*, 5(3):269–287, 1983.
[45] J. Shi, L. Dhulipala, and J. Shun. Parallel clique counting and peeling algorithms. In *SIAM Conf. on Applied and Computational Discrete Algorithms (ACDA21)*, pages 135–146. SIAM, 2021.
[46] K. Shin, T. Eliassi-Rad, and C. Faloutsos. Corescope: Graph mining using k-core analysis—patterns, anomalies and algorithms. In *2016 IEEE 16th international Conf. on data mining (ICDM)*, pages 469–478. IEEE, 2016.
[47] K. Shin, B. Hooi, J. Kim, and C. Faloutsos. Densealert: Incremental dense-subtensor detection in tensor streams. In *Proceedings of the 23rd ACM SIGKDD International Conf. on Knowledge Discovery and Data Mining*, pages 1057–1066, 2017.
[48] P. Sukprasert, Q. C. Liu, L. Dhulipala, and J. Shun. Practical parallel algorithms for near-optimal densest subgraphs on massive graphs. In *2024 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 59–73. SIAM, 2024.
[49] B. Sun, M. Danisch, T. H. Chan, and M. Sozio. Kclist++: A simple algorithm for finding k-clique densest subgraphs in large graphs. *Proceedings of the VLDB Endowment (PVLDB)*, 2020.
[50] C. Tsourakakis. The k-clique densest subgraph problem. In *Proceedings of the 24th international Conf. on world wide web*, pages 1122–1132, 2015.
[51] S. Yao, Y. Li, S. Sun, J. Jiang, and B. He. ublade: Efficient batch processing for uncertainty graph queries. *Proceedings of the ACM on Management of Data*, 2(3):1–24, 2024.
[52] C. Ye, Y. Li, B. He, Z. Li, and J. Sun. Gpu-accelerated graph label propagation for real-time fraud detection. In *Proceedings of the 2021 International Conf. on Management of Data*, pages 2348–2356, 2021.

Received October 2024; revised January 2025; accepted February 2025