

# Scaling Up Density Decomposition at Billion-Scale Graphs

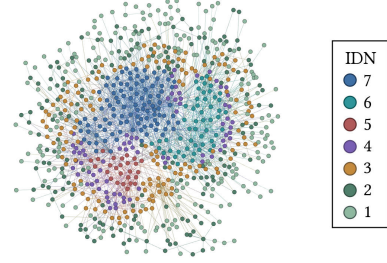
## ABSTRACT

Density decomposition characterizes the multi-level dense structure of large networks and supports a wide range of graph mining applications. Given a graph  $G = (V, E)$ , it assigns each vertex an integral dense number (IDN) and produces a nested sequence of layers  $D_0 \supseteq D_1 \supseteq \dots \supseteq D_p$  that capture increasingly dense and structurally important regions of the graph. However, existing algorithms for computing density decomposition are computationally expensive. The state-of-the-art divide-and-conquer algorithm BinaryDC runs in  $O(\log p \cdot |E|^{1.5})$  time, but its binary-search-based pivot selection incurs multiple max-flow computations per divide step, making it prohibitively slow on massive graphs. To address this problem, we first propose MeanDC, which replaces binary-search-based pivot selection with a mean-based strategy, significantly reducing the number of max-flow while preserving the same asymptotic complexity. We then propose a novel *core-prepartition* technique and develop CoreDC, which leverages the  $k$ -core hierarchy to pre-split the graph into  $O(\log p)$  disjoint regions and performs localized divide-and-conquer within each region, greatly shrinking the size of flow networks. For parallel computation, we develop *heat-extraction* technique via a lightweight, edge-local heat-diffusion iteration that is highly parallelizable and can extract many layers without any max-flow calls. Building on this idea, we design HeatDC and further combine heat-extraction with core-prepartition into our fastest parallel algorithm CoreHeatDC. Experiments on 10 real-world graphs with up to 1.81 billion edges demonstrate that our algorithms consistently outperform existing approaches, and that CoreHeatDC is able to compute density decomposition of billion-scale graphs within a few hundred seconds.

## 1 INTRODUCTION

Graph decomposition is a fundamental tool for understanding the structural organization of large networks, with applications in community detection [1, 17, 25, 33, 42, 55], fraud detection [6], graph querying [19, 32], and graph visualization [2, 61]. A wide range of decomposition models have been studied, including degree-based core decomposition [4, 26, 27, 37, 43], triangle-based truss decomposition [20, 29, 52], and clique- or nucleus-based methods [44–46, 48]. While effective in capturing certain notions of cohesiveness, these approaches are largely driven by local structural constraints and may fail to fully reflect edge density.

Recently, *density decomposition* has emerged as a powerful alternative for revealing the multi-level dense organization of real-world networks [11, 12, 18, 54, 56, 59, 60]. Given a graph  $G = (V, E)$ , it produces a nested sequence of vertex-induced subgraphs  $D_0 = V \supseteq D_1 \supseteq \dots \supseteq D_p \supseteq D_{p+1} = \emptyset$ , where  $p$  is the largest integer such that  $D_p$  is non-empty and higher layers correspond to increasingly dense and structurally important regions of the graph (Definition 1). For a vertex  $u \in V$ , its *integral dense number* (IDN) is defined as the largest  $k$  such that  $u \in D_k$ , which captures the density level at which the vertex resides in the graph. Figure 1 illustrates the density decomposition of a real network Wiki-Vote<sup>1</sup>. As seen, vertices with high IDN values (e.g., IDN = 5, 6, 7) aggregate into several cohesive communities, whereas vertices with low IDN values (e.g., IDN = 1, 2) are pushed to the periphery. Overall, density decomposition provides an informative hierarchical view of graph density.



Vertices are colored by IDN;  $D_k$  contains all vertices with IDN  $\geq k$ .

**Figure 1: Density decomposition of Wiki-Vote network.**

Despite its usefulness, computing density decomposition remains highly challenging at scale. Given an integer  $k$ , computing a single layer  $D_k$  requires one max-flow invocation with time complexity  $O(|E|^{1.5})$  [8, 56, 60]. A straightforward algorithm, IncrFlow [56, 60], computes all layers by invoking the max-flow procedure for every  $k$  from 0 to  $p$ , incurring a time complexity of  $O(p \cdot |E|^{1.5})$ . For real-world large graphs, where the number of layers  $p$  is often in the hundreds or even thousands [58], IncrFlow is difficult to scale.

The state-of-the-art (SOTA) algorithm BinaryDC [56, 60] reduces this cost to  $O(\log p \cdot |E|^{1.5})$  by adopting a divide-and-conquer framework together with a *localized max-flow* technique. Specifically, if two layers  $D_k^{lb}$  and  $D_k^{ub}$  are known such that  $D_k^{ub} \subseteq D_k \subseteq D_k^{lb}$ , then  $D_k$  can be computed by running max-flow on the subgraph  $D_k^{lb} \setminus D_k^{ub}$  with a local time complexity of  $O(|E(D_k^{lb} \setminus D_k^{ub})|^{1.5})$ . Using this idea, BinaryDC recursively selects a pivot  $k \in (l, u)$  between two known layers  $D_l$  and  $D_u$ , computes  $D_k$  on  $D_l \setminus D_u$ , and partitions the problem into smaller subranges. However, to satisfy the halving requirement of divide-and-conquer analysis, BinaryDC relies on a binary search to locate a pivot that strictly bisects the graph, requiring  $O(\log p)$  max-flow computations per divide step. On massive graphs (e.g., billion-edge graphs), this overhead becomes a major performance bottleneck.

To address this inefficiency, we propose MeanDC (Algorithm 4), which replaces the binary-search-based pivot selection in BinaryDC requiring  $O(\log p)$  max-flow computations, with a single arithmetic-mean choice that uses only one max-flow computation. Although this choice no longer guarantees strict halving, we show that the total cost of all local max-flow computations at the same recursion depth is bounded by  $O(|E|^{1.5})$  (Lemma 1). Since the recursion depth is at most  $O(\log p)$ , MeanDC preserves the same overall time complexity  $O(\log p \cdot |E|^{1.5})$  as BinaryDC, while enabling much faster graph partitioning and most max-flow computations to be executed on much smaller subgraphs.

Despite this improvement, the first divide step of MeanDC still requires a max-flow computation on the entire graph. To eliminate this expensive operation, we exploit the relationship between density decomposition and *core decomposition* [4, 35, 37, 39, 43]. By the Sandwich Theorem, the two satisfy the relation  $C_{2k} \subseteq D_k \subseteq C_k$  [56], where  $C_k$  denotes the  $k$ -core of the graph, i.e., the largest subgraph in which every vertex has degree at least  $k$ . Based on this result, we develop a novel *core-prepartition* technique and propose the algorithm CoreDC (Algorithm 5). The key idea is to use the  $k$ -core hierarchy to pre-split the graph into  $O(\log p)$  disjoint regions, and then perform localized max-flow and divide-and-conquer within each region. In the first partitioning step, MeanDC requires

<sup>1</sup>Dataset source: <https://networkrepository.com/soc-wiki-Vote.php>.

an  $O(|E|^{1.5})$ -time max-flow computation to obtain a single partition, whereas CoreDC uses the same  $O(|E|^{1.5})$  total time to produce  $O(\log p)$  partitions, substantially reducing the subgraph scale and the cost of the subsequent recursion.

In the sequential setting, CoreDC is already highly efficient for computing density decomposition. However, for large-scale graphs, even this improved sequential performance remains computationally expensive, motivating the use of multi-threading for acceleration. Although CoreDC can be parallelized by assigning the two branches of its divide-and-conquer procedure to different threads, the resulting task granularity remains relatively coarse, and the parallel speedup tends to saturate at around 8 threads (Exp-2). To improve parallel efficiency, we propose a novel *heat-extraction* technique and present the HeatDC algorithm (Algorithm 6). Specifically, the heat-extraction procedure runs an iterative heat-diffusion process that simulates heat conduction in the real world: each edge is treated as a heat-producing unit, and heat is repeatedly transferred toward colder vertices. As the number of iterations  $t$  increases, the heat value  $h^t(u)$  of each vertex  $u \in V$  grows, and we prove that  $\lfloor h^t(u)/t \rfloor$  converges to the IDN of  $u$ . After a sufficient number of iterations yields accurate IDN approximations, the algorithm heuristically extracts multiple density decomposition layers from the heat values, which can directly recover up to 89% of all layers across our experimental datasets (Exp-5). HeatDC then applies a divide-and-conquer procedure to compute the remaining layers. Since the heat-diffusion process is local and highly parallelizable, HeatDC achieves excellent parallel speedup.

Finally, by integrating core-prepartition and heat-extraction, we propose the algorithm CoreHeatDC. These two techniques are complementary: core-prepartition provides predictable and balanced coarse partitions, while heat-extraction supplies additional fine-grained layers and fully utilizes idle threads. CoreHeatDC executes both procedures concurrently, takes the union of all precomputed layers, and completes the remaining layers via divide-and-conquer. In our experiments, CoreHeatDC computes the density decomposition of billion-edge graphs within a few hundred seconds, making density decomposition scalable to billion-scale graphs.

We summarize our main contributions as follows:

**Simple yet efficient divide-and-conquer.** We propose MeanDC, which simplifies the pivot selection in BinaryDC from a costly binary search requiring  $O(\log p)$  max-flow computations to a single arithmetic-mean choice with only one max-flow computation. Despite relaxing strict halving, MeanDC preserves the same asymptotic time complexity  $O(\log p \cdot |E|^{1.5})$  while significantly accelerating graph shrinking in practice.

**Core-prepartition for more localized max-flow computation.** We introduce a core-prepartition technique based on the Sandwich Theorem and develop CoreDC, which leverages the  $k$ -core hierarchy to pre-split the graph into  $O(\log p)$  disjoint regions before performing localized max-flow and divide-and-conquer. This approach substantially reduces the size of subgraphs for subsequent computations, leading to significant efficiency improvements.

**Parallel heat-extraction technique for billion-scale graphs.** In parallel settings, we propose a novel heat-extraction technique, which employs a parallel-friendly heat-diffusion iteration to approximate IDN values and heuristically extract multiple density layers without max-flow computations. We formally prove that our heat-diffusion converges to the exact IDN given a sufficient number of iterations. By combining core-prepartition and heat-extraction, we develop CoreHeatDC, which achieves the best practical performance and scales density decomposition to billion-scale graphs.

**Extensive experiments.** We conduct extensive experiments on 10 real-world graphs from diverse domains, with up to 68.7M vertices and 1.81B edges. Our results show that: (1) *In the sequential setting*, our algorithms consistently outperform the baselines, with our CoreDC achieving the best overall performance. For example, on the largest dataset SKAll, BinaryDC (SOTA) and CoreDC take 9,595s and 3,523s, respectively, yielding a  $2.7\times$  speedup. (2) *In the parallel setting*, CoreHeatDC delivers the best performance across all datasets with 32 threads and completes billion-edge graphs within a few hundred seconds. For instance, on SKAll with 1.81B edges, CoreHeatDC finishes in 675s and outperforms the SOTA algorithm by  $9.4\times$ . Moreover, CoreHeatDC achieves strong parallel speedup, e.g., a  $17.7\times$  speedup on DBpedia with 32 threads. In terms of memory usage, the baseline algorithm IncrFlow exceeds 256 GB on SKAll, whereas our proposed algorithms use only linear  $O(|E|)$  memory and never consume more than 60 GB in our experiments. (3) *Effectiveness of core-prepartition and heat-extraction.* In the divide-and-conquer procedure, both techniques significantly reduce the average runtime and the average number of edges per recursive call by transforming large global max-flow instances into many smaller, localized ones (Exp-4). More importantly, their combination produces more partitions with fewer edges per partition, enabling the divide-and-conquer process to start with a larger number of independent subproblems (Exp-5). These results confirm the effectiveness of our techniques in improving efficiency.

**Reproducibility and full version paper.** The source code and full version of this paper can be found at <https://anonymous.4open.science/r/ParDD>.

## 2 PRELIMINARIES

In this paper, we consider an unweighted and undirected graph  $G = (V, E)$ , where  $V$  and  $E$  denote the vertex set and the edge set, respectively. For a vertex  $u \in V$ , its degree in  $G$  is denoted by  $d(u, G)$ , or simply  $d(u)$  when the context is clear. Given a vertex subset  $S \subseteq V$ , let  $E(S)$  denote the set of edges induced by  $S$ , i.e.,  $E(S) = \{(u, v) \in E \mid u \in S, v \in S\}$ , and let  $G[S] = (S, E(S))$  denote the corresponding induced subgraph. With these notations in place, we now introduce the definition of *density decomposition*.

**DEFINITION 1. (Density Decomposition)** [11, 56, 60] Given a graph  $G = (V, E)$ , for any non-negative integer  $k$ , the vertex set  $D_k$  is defined as the subset of vertices satisfying:

- (1) **Internally dense:** For any non-empty subset  $S \subseteq D_k$ , it holds that  $|E(D_k)| - |E(D_k \setminus S)| > (k-1)|S|$ .
- (2) **Externally sparse:** For any non-empty subset  $T \subseteq (V \setminus D_k)$ , it holds that  $|E(D_k \cup T)| - |E(D_k)| \leq (k-1)|T|$ .

Let  $p$  be the largest integer such that  $D_p \neq \emptyset$ . Then, the density decomposition of  $G$  is defined as  $\mathcal{D} = \{D_0, D_1, \dots, D_p\}$ .

Intuitively, the above definition characterizes each layer  $D_k$  from two complementary perspectives. Removing any subset  $S \subseteq D_k$  leads to a substantial loss of edges (*internally dense*). In contrast, adding any subset  $T \subseteq (V \setminus D_k)$  introduces only a limited number of additional edges (*externally sparse*). Together, each layer represents a densely connected region that is cohesive internally and well separated from the rest of the graph. We next summarize several fundamental properties of density decomposition.

**THEOREM 1.** [11] Given a graph  $G = (V, E)$ , the density decomposition  $\mathcal{D} = \{D_0, D_1, \dots, D_p\}$  satisfies the following properties:

- (1) **Uniqueness property:** For a fixed graph  $G$ , each layer  $D_k$  is uniquely determined.

- (2) **Hierarchy property:** The layers form a nested hierarchy such that  $D_0 = V \supseteq D_1 \supseteq \dots \supseteq D_p \supseteq D_{p+1} = \emptyset$ .

According to these properties, the density decomposition divides the graph into a hierarchical structure of nested dense regions. We next introduce the *integral dense number (IDN)* to quantify the highest density level to which each vertex belongs. Intuitively, a larger IDN indicates that a vertex lies in a denser and more central region of the graph.

**DEFINITION 2. (Integral Dense Number, IDN)** [56, 60] Given the density decomposition  $\{D_0, D_1, \dots, D_p\}$  of a graph  $G = (V, E)$ , the integral dense number of a vertex  $u \in V$ , denoted by  $\tilde{r}(u)$ , is defined as the maximum integer such that  $u \in D_{\tilde{r}(u)}$  and  $u \notin D_{\tilde{r}(u)+1}$ .

**EXAMPLE 1.** Figure 1 illustrates the density decomposition of the real-world social network Wiki-Vote. By definition, the  $k$ -th layer  $D_k$  contains all vertices whose IDN values are at least  $k$ ; for example, the layer  $D_5$  includes vertices with IDN values 5, 6, and 7. The decomposition identifies a densely connected community as the top layer  $D_7$ , indicating that  $p = 7$  and  $D_k = \emptyset$  for all  $k > p$ . Surrounding  $D_7$ , two coherent but less dense communities are revealed, including  $D_6 \setminus D_7$  and  $D_5 \setminus D_6$ , corresponding to vertices with IDN values 6 and 5, respectively. In contrast, vertices located at the periphery of the graph are assigned low IDN values (e.g., IDN 1 or 2), reflecting their sparse connectivity and marginal structural roles. Overall, this example demonstrates how density decomposition captures the density structure of real-world networks.

**The  $k$ -core and degeneracy.** Another hierarchical decomposition closely related to density decomposition is the *core decomposition* [4]. It shares strong theoretical connections with density decomposition and will later be leveraged to optimize our algorithms. We next formally introduce this concept.

**DEFINITION 3. ( $k$ -Core and Core Decomposition)** [4] Given a graph  $G = (V, E)$  and a non-negative integer  $k$ , the  $k$ -core of  $G$  is the vertex set  $C_k$  such that  $G[C_k]$  is the maximal subgraph in which every vertex has degree at least  $k$ . The degeneracy  $\delta$  of  $G$  is defined as the largest integer for which  $C_\delta$  is non-empty. The core decomposition of  $G$  is then defined as  $\mathcal{C} = \{C_0, C_1, \dots, C_\delta\}$ .

The following theorem establishes the nesting relationships between density decomposition and core decomposition.

**THEOREM 2. (The Sandwich Theorem)** [56, 60] Given a graph  $G$  and a non-negative integer  $k$ , it holds that  $C_{2k} \subseteq D_k \subseteq C_k$ , and thus  $p \leq \delta \leq 2p$ .

Since the core decomposition can be computed in  $O(|E|)$  time [4], Theorem 2 enables us to efficiently obtain a 2-approximation of  $p$  (i.e., the degeneracy  $\delta$ ) and to roughly localize each density layer  $D_k$  between  $C_k$  and  $C_{2k}$  in linear time. However, as shown in prior work [56, 60], density decomposition is driven by subgraph density rather than vertex degree alone, and thus captures dense structures more precisely than core decomposition.

With these preliminaries, we formulate our problem as follows.

**Problem definition.** Given an undirected graph  $G = (V, E)$ , the goal is to efficiently compute its density decomposition, i.e., to compute the IDN for all vertices in  $V$ . We address this problem in both sequential and parallel settings, aiming to develop algorithms that are both theoretically efficient and practically scalable.

### 3 EXISTING ALGORITHMS AND LIMITATIONS

**Computing a single layer of the density decomposition.** A fundamental subroutine for density decomposition is computing a

---

#### Algorithm 1: GetDk( $G, k$ ) [56, 60]

---

**Input:** A graph  $G = (V, E)$  and an integer  $k$ .  
**Output:** The  $k$ -th layer  $D_k$  in density decomposition.

- 1 Initialize a flow network  $N = (V \cup \{s, t\}, \vec{E}, c)$ , where  $\vec{E}$  is a set of directed arcs and  $c$  is the capacity function;
- 2 **foreach**  $(u, v) \in E$  **do**  $\vec{E} \leftarrow \vec{E} \cup (u, v)$ ,  $c(u, v) \leftarrow 1$ ;
- 3 For nodes in  $V$ , compute their indegree  $\vec{d}(\cdot)$  in  $\vec{E}$ ;
- 4 **foreach**  $u \in V$  **do**
  - 5   **if**  $\vec{d}(u) < k - 1$  **then**  $\vec{E} \leftarrow \vec{E} \cup (s, u)$ ,  $c(s, u) \leftarrow (k - 1) - \vec{d}(u)$ ;
  - 6   **if**  $\vec{d}(u) > k - 1$  **then**  $\vec{E} \leftarrow \vec{E} \cup (u, t)$ ,  $c(u, t) \leftarrow \vec{d}(u) - (k - 1)$ ;
- 7 Compute the max flow for  $N$  and let  $(S, T)$  be the minimum  $s$ - $t$  cut;
- 8  $D_k \leftarrow T \setminus \{t\}$ ;
- 9 **return**  $D_k$ ;

---



---

#### Algorithm 2: LocalGetDk( $G, k, D_k^{lb}, D_k^{ub}$ ) [56, 60]

---

**Input:** A graph  $G = (V, E)$ , an integer  $k$  and two vertex set  $D_k^{lb}$  and  $D_k^{ub}$ , satisfying  $D_k^{ub} \subseteq D_k \subseteq D_k^{lb}$ .  
**Output:** The  $k$ -th layer  $D_k$  in density decomposition.

- 1 Initialize a flow network  $N = ((D_k^{lb} \setminus D_k^{ub}) \cup \{s, t\}, \vec{E}, c)$ , where  $\vec{E}$  is a set of directed arcs and  $c$  is the capacity function;
- 2 **foreach**  $(u, v) \in E(D_k^{lb} \setminus D_k^{ub})$  **do**  $\vec{E} \leftarrow \vec{E} \cup (u, v)$ ,  $c(u, v) \leftarrow 1$ ;
- 3 For nodes in  $(D_k^{lb} \setminus D_k^{ub})$ , compute their indegree  $\vec{d}(\cdot)$  in  $\vec{E}$ ;
- 4 **foreach**  $(u, v), u \in (D_k^{lb} \setminus D_k^{ub}), v \in D_k^{ub}$  **do**  $\vec{d}(u) \leftarrow \vec{d}(u) + 1$ ;
- 5 **foreach**  $u \in (D_k^{lb} \setminus D_k^{ub})$  **do**
  - 6   **if**  $\vec{d}(u) < k - 1$  **then**  $\vec{E} \leftarrow \vec{E} \cup (s, u)$ ,  $c(s, u) \leftarrow (k - 1) - \vec{d}(u)$ ;
  - 7   **if**  $\vec{d}(u) > k - 1$  **then**  $\vec{E} \leftarrow \vec{E} \cup (u, t)$ ,  $c(u, t) \leftarrow \vec{d}(u) - (k - 1)$ ;
- 8 Compute the max flow for  $N$  and let  $(S, T)$  be the minimum  $s$ - $t$  cut;
- 9  $D_k \leftarrow (T \setminus \{t\}) \cup D_k^{ub}$ ;
- 10 **return**  $D_k$ ;

---

target layer  $G_k$  for a given  $k$ . This can be accomplished using the GetDk algorithm [7, 56, 60] (Algorithm 1). The algorithm constructs a flow network and computes its max-flow, where the resulting minimum  $s$ - $t$  cut exactly yields  $D_k$ . When implemented using Dinic's max-flow algorithm [22], GetDk runs in  $O(|E|^{1.5})$  time [8, 56, 60]. A more general variant of GetDk, called LocalGetDk, was proposed in [56, 60] (Algorithm 2). In addition to the input graph and target parameter, LocalGetDk takes two vertex subsets  $D_k^{lb}$  and  $D_k^{ub}$  as input, which must satisfy  $D_k^{ub} \subseteq D_k \subseteq D_k^{lb}$ . Under this constraint, LocalGetDk restricts the max-flow computation to the local subgraph induced by  $D_k^{lb} \setminus D_k^{ub}$ , thereby computing  $D_k$  with a local time complexity of  $O(|E(D_k^{lb} \setminus D_k^{ub})|^{1.5})$  and a space complexity of  $O(|E(D_k^{lb} \setminus D_k^{ub})|)$  [56, 60]. As an illustrative example, if the layers  $D_5$  and  $D_{10}$  have already been computed, we can obtain  $D_7$  by invoking LocalGetDk( $G, 7, D_5, D_{10}$ ), which runs in local  $O(|E(D_5 \setminus D_{10})|^{1.5})$  time.

**Computing the density decomposition.** With GetDk available, a straightforward approach to compute the density decomposition is to start from  $k = 0$  and iteratively invoke GetDk to obtain  $D_k$  until  $D_k = \emptyset$ ; this yields the IncrFlow algorithm [56, 60]. While simple, IncrFlow performs  $p + 2$  max-flow computations and thus has a total running time of  $O(p \cdot |E|^{1.5})$ , making it difficult to scale to large graphs.

To address the inefficiency of IncrFlow, where each max-flow computation is performed on the entire graph, the BinaryDC algorithm [56, 60] adopts a divide-and-conquer strategy that recursively partitions the graph so that most max-flow computations can be performed locally using LocalGetDk [56, 60]. Specifically, as

---

**Algorithm 3: BinaryDC( $G$ )** [56, 60]

---

**Input:** A graph  $G = (V, E)$ .  
**Output:** The density decomposition  $\mathcal{D}$  of  $G$ .

- 1 Compute the degeneracy  $\delta$  as a 2-approximation of  $p$ ;
- 2  $D_0 \leftarrow V, D_{\delta+1} \leftarrow \emptyset, \mathcal{D} \leftarrow \{D_0\}$ ;
- 3  $\text{Divide}(D_0, D_{\delta+1})$ ;
- 4 **return**  $\mathcal{D}$ ;

5 **Procedure**  $\text{Divide}(D_l, D_u)$

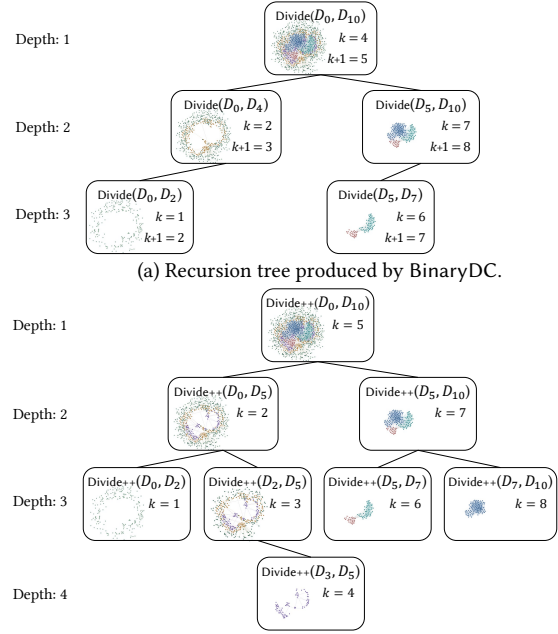
- 6 **if**  $u - l \leq 1$  or  $D_l \setminus D_u = \emptyset$  **then return**;
- 7  $k_u \leftarrow u, k_l \leftarrow l$ ;
- 8 **while**  $k_u > k_l$  **do** // Binary search to find the pivot  $k$ 
  - 9  $k_m \leftarrow \lfloor (k_u + k_l) / 2 \rfloor$ ;
  - 10  $D_{k_m} \leftarrow \text{LocalGetDk}(G, k_m, D_l, D_u)$ ;
  - 11 **if**  $|E(R_l \setminus R_{k_m})| < |E(R_l \setminus R_u)| / 2$  **then**  $k_l \leftarrow k_m + 1$ ;
  - 12 **else**  $k_u \leftarrow k_m$ ;
- 13  $k \leftarrow k_l$ ;
- 14  $D_k \leftarrow \text{LocalGetDk}(G, k, D_l, D_u)$ ;  $\mathcal{D} \leftarrow \mathcal{D} \cup D_k$ ;
- 15  $\text{Divide}(D_l, D_k)$ ;
- 16  $D_{k+1} \leftarrow \text{LocalGetDk}(G, k + 1, D_l, D_u)$ ;  $\mathcal{D} \leftarrow \mathcal{D} \cup D_{k+1}$ ;
- 17  $\text{Divide}(D_{k+1}, D_u)$ ;

---

shown in Algorithm 3, BinaryDC first computes an upper bound of  $p$  using its 2-approximation degeneracy  $\delta$ . The algorithm then initializes two boundary layers  $D_0 = V$  and  $D_{\delta+1} = \emptyset$ , and invokes the recursive procedure  $\text{Divide}(D_0, D_{\delta+1})$  (lines 2–3). In each call to  $\text{Divide}$ , the algorithm performs a binary search to identify a pivot  $k$  such that both  $|E(D_l \setminus D_k)|$  and  $|E(D_{k+1} \setminus D_u)|$  are no larger than  $|E(D_l \setminus D_u)| / 2$  (lines 7–13). It then computes  $D_k$  and  $D_{k+1}$  and continues the recursion on the two subranges  $(l, k)$  and  $(k + 1, u)$  (lines 14–17). This binary search ensures that, at each recursion level, the problem size is reduced by a factor of two, thereby satisfying the prerequisite of the divide-and-conquer Master Theorem [5]. As a result, the overall time complexity is reduced from  $\text{IncrFlow}$ 's  $O(p \cdot |E|^{1.5})$  to  $O(\log p \cdot |E|^{1.5})$  [56, 60].

**EXAMPLE 2.** The execution of BinaryDC on the example graph in Figure 1 is illustrated in Figure 2a. The algorithm first computes  $\delta + 1 = 10$  and initiates the recursion at depth 1 by invoking  $\text{Divide}(D_0, D_{10})$ . In this call, BinaryDC identifies a pivot  $k = 4$  via binary search, which requires four search steps and consequently four max-flow computations. The algorithm then performs two additional max-flow computations to obtain the boundary layers  $D_4$  and  $D_5$ , after which the graph is partitioned into two subgraphs,  $D_0 \setminus D_4$  and  $D_5 \setminus D_{10}$ . Subsequently, BinaryDC proceeds with two recursive  $\text{Divide}$  calls at depth 2 on these subgraphs and continues recursively until all density layers are computed.

**Limitations.** Despite the reduction in asymptotic time complexity, BinaryDC incurs substantial practical overhead due to its binary search for the pivot: each recursive call invokes multiple max-flow computations solely to identify an appropriate split point. In particular, already at the first recursion level, BinaryDC requires  $\Theta(\log p)$  full-graph max-flow computations, each taking  $O(|E|^{1.5})$  time, a prohibitive expense on massive graphs. For example, in Exp-4 of our experiments, the runtime spent at depth 1 of BinaryDC exceeds the total runtime of all deeper recursion depths combined, indicating that a significant portion of the computation is consumed by pivot searching with max-flow. These limitations motivate our study of more efficient sequential and parallel algorithms that reduce unnecessary max-flow calls and better exploit locality and parallelism.



**Figure 2: Recursion trees of BinaryDC and MeanDC on the example graph in Figure 1.**

---

**Algorithm 4: MeanDC( $G$ )**


---

**Input:** A graph  $G = (V, E)$ .  
**Output:** The density decomposition  $\mathcal{D}$  of  $G$ .

- 1 Compute the degeneracy  $\delta$  as a 2-approximation of  $p$ ;
- 2  $D_0 \leftarrow V, D_{\delta+1} \leftarrow \emptyset, \mathcal{D} \leftarrow \{D_0\}$ ;
- 3  $\text{Divide}++(D_0, D_{\delta+1})$ ;
- 4 **return**  $\mathcal{D}$ ;

5 **Procedure**  $\text{Divide}++(D_l, D_u)$

- 6 **if**  $u - l \leq 1$  or  $D_l \setminus D_u = \emptyset$  **then return**;
- 7  $k \leftarrow \lfloor \frac{l+u}{2} \rfloor$ ; // Use arithmetic mean instead of binary search
- 8  $D_k \leftarrow \text{LocalGetDk}(G, k, D_l, D_u)$ ;  $\mathcal{D} \leftarrow \mathcal{D} \cup D_k$ ;
- 9  $\text{Divide}++(D_l, D_k)$ ;
- 10  $\text{Divide}++(D_k, D_u)$ ;

---

## 4 NEW DIVIDE-AND-CONQUER ALGORITHM

In this section, we show that the binary search used in BinaryDC is unnecessary: simply setting  $k$  to the arithmetic mean  $\lfloor (l + u) / 2 \rfloor$  suffices to guarantee the same time complexity of  $O(\log p \cdot |E|^{1.5})$ , while resulting in a more intuitive and more efficient algorithmic workflow. Based on this simplified divide-and-conquer strategy, we present the MeanDC algorithm in Algorithm 4. It employs the procedure  $\text{Divide}++$ , which replaces the complicated binary search with a single operation  $k \leftarrow \lfloor (l + u) / 2 \rfloor$  (line 7). The algorithm then computes  $D_k$  and recursively processes the two resulting subranges using  $\text{Divide}++$ .

**EXAMPLE 3.** The execution of MeanDC on the example graph in Figure 1 is illustrated in Figure 2b. At depth 1, unlike BinaryDC, which incurs multiple max-flow computations to identify a pivot, MeanDC directly sets  $k = 5$ , which is the arithmetic mean of 0 and 10. The algorithm then computes  $D_5$ , partitions the graph into two subgraphs  $D_0 \setminus D_5$  and  $D_5 \setminus D_{10}$ , and proceeds with deeper recursion at depth 2. Although the recursion tree of MeanDC is deeper than that of BinaryDC, this design shifts many max-flow computations to smaller

subgraphs at deeper recursion levels. In contrast, BinaryDC performs most of its max-flow computations at shallow recursion depths, where the subgraphs are substantially larger. As a result, MeanDC achieves significantly better practical efficiency as shown in our experiments.

Below, we prove that the MeanDC algorithm preserves the time complexity of  $O(\log p \cdot |E|^{1.5})$  for density decomposition. We begin by presenting the following lemma.

**LEMMA 1.** *Consider the recursion tree produced by the Divide++ subroutine in MeanDC, where each node in the tree is represented by a pair  $(D_l, D_u)$ . For any depth  $h \geq 0$ , let  $\mathcal{P}_h$  denote the set of such pairs at depth  $h$  in the recursion tree. The total time complexity of all LocalGetDk calls at recursion depth  $h$  is  $O(|E|^{1.5})$ .*

**PROOF.** Each recursive step splits the current region  $D_l \setminus D_u$  into two disjoint subregions, namely  $D_l \setminus D_k$  and  $D_k \setminus D_u$ . Consequently, the regions corresponding to all pairs in  $\mathcal{P}_h$  are pairwise disjoint, implying that  $\sum_{(D_l, D_u) \in \mathcal{P}_h} |E(D_l \setminus D_u)| \leq |E|$ . For  $\alpha \geq 1$ , the function  $x \mapsto x^\alpha$  is convex with  $f(0) = 0$ , which implies super-additivity on  $\mathbb{R}_{\geq 0}$ , i.e.,  $x^\alpha + y^\alpha \leq (x + y)^\alpha$  for all  $x, y \geq 0$ . Applying this with  $\alpha = 1.5$  and iterating over the disjoint edge counts yields  $\sum_{(D_l, D_u) \in \mathcal{P}_h} |E(D_l \setminus D_u)|^{1.5} \leq (\sum_{(D_l, D_u) \in \mathcal{P}_h} |E(D_l \setminus D_u)|)^{1.5} \leq |E|^{1.5}$ . Since each invocation of LocalGetDk requires  $O(|E(D_l \setminus D_u)|^{1.5})$  time, the total time complexity of all LocalGetDk calls at recursion depth  $h$  is  $O(|E|^{1.5})$ .  $\square$

By Lemma 1, we establish Theorem 3.

**THEOREM 3.** *The MeanDC algorithm correctly returns the density decomposition  $\mathcal{D}$ , with a time complexity of  $O(\log p \cdot |E|^{1.5})$  and a space complexity of  $O(|E|)$ .*

**PROOF.** By the Sandwich Theorem, we have  $\delta + 1 > p \Rightarrow D_{\delta+1} = \emptyset$ . Together with the two boundary layers  $D_0$  and  $D_{\delta+1}$  introduced in line 2, all non-empty layers of the density decomposition, namely  $D_0, D_1, \dots, D_p$ , are covered. Next, the procedure Divide++( $D_l, D_u$ ) can be viewed as follows: given two known layers  $D_l$  and  $D_u$ , it computes all intermediate layers  $D_{l+1}, D_{l+2}, \dots, D_{u-1}$ . This property follows straightforwardly by induction. Therefore, the call Divide++( $D_0, D_{\delta+1}$ ) in line 3 is guaranteed to collect all layers into  $\mathcal{D}$ , which establishes the correctness of MeanDC.

For time complexity, the operation  $k \leftarrow \lfloor \frac{l+u}{2} \rfloor$  guarantees that each recursive step halves the index range length  $(u - l)$ . Starting from an initial range of length  $\delta + 1$ , the recursion depth is thus bounded by  $O(\log \delta) = O(\log p)$ , since  $\delta \leq 2p$  according to the Sandwich Theorem (Theorem 2). By Lemma 1, the total cost incurred at any fixed recursion depth is  $O(|E|^{1.5})$ . Multiplying by the number of recursion levels yields an overall time complexity of  $O(\log p \cdot |E|^{1.5})$ . For the space complexity, each call to LocalGetDk requires  $O(|E(D_l \setminus D_u)|)$  space. Since the recursion operates on disjoint regions, the space complexity of MeanDC is  $O(|E|)$ .  $\square$

**Comparison between BinaryDC and MeanDC.** From a theoretical perspective, removing the binary search strategy for selecting the pivot  $k$  in MeanDC does not weaken the algorithm compared to BinaryDC. To the best of our knowledge, there is no class of graphs for which MeanDC exhibits a worse asymptotic time complexity than BinaryDC. On the contrary, there exist scenarios in which MeanDC is strictly stronger. We intuitively consider the following two extreme examples.

• **Case 1: Balanced layers.** All layers contain approximately the same number of edges, i.e.,  $|E(D_k \setminus D_{k+1})|$  is nearly the same for all  $k \in [0, p]$ . In this case, the arithmetic mean pivot selection in

#### Algorithm 5: CoreDC( $G$ )

---

**Input:** A graph  $G = (V, E)$ .  
**Output:** The density decomposition  $\mathcal{D}$  of  $G$ .

```

1  $\mathcal{D}_C \leftarrow \text{CorePrepartition}(G)$ ;
2  $\mathcal{D} \leftarrow \mathcal{D}_C$ ;
3 for  $(l, u) \in \text{Gap}(\mathcal{D}_C)$  do // Compute remaining layers
4    $\text{Divide++}(D_l, D_u)$ ;
5 return  $\mathcal{D}$ ;

6 Procedure CorePrepartition( $G$ )
7   Compute the core decomposition  $\mathcal{C} \leftarrow \{C_0, C_1, \dots, C_\delta\}$ ;
8    $D_0 \leftarrow C_0, D_1 \leftarrow C_1, D_{\lfloor \log_2 \delta \rfloor + 1} \leftarrow \emptyset, \mathcal{D}_C \leftarrow \{D_0, D_1, D_{\lfloor \log_2 \delta \rfloor + 1}\}$ ;
9   for  $i = 1, \dots, \lfloor \log_2 \delta \rfloor$  do
10    if  $2^{i+1} > \delta$  then  $C_{2^{i+1}} \leftarrow \emptyset$ ;
11     $D_{2^i} \leftarrow \text{LocalGetDk}(G, 2^i, C_{2^i}, C_{2^{i+1}}), \mathcal{D}_C \leftarrow \mathcal{D}_C \cup \{D_{2^i}\}$ ;
12  return  $\mathcal{D}_C$ ;
```

---

MeanDC naturally ensures that each recursive step reduces the problem size by roughly a factor of two, thereby satisfying the requirement of the divide-and-conquer Master Theorem [5]. Consequently, the worst-case time complexity reduces to  $O(|E|^{1.5})$ . In contrast, BinaryDC performs  $\Theta(\log p)$  global max-flow computations on the entire graph at the first recursion depth, leading to an overall complexity no better than  $O(\log p \cdot |E|^{1.5})$ . This demonstrates that for such balanced graphs, MeanDC is strictly more efficient than BinaryDC.

• **Case 2: Single-layer concentration.** All edges are concentrated in a single layer, i.e., all vertices have  $\tilde{r}(u) = p$ . In this scenario, the binary search strategy in BinaryDC and the arithmetic mean selection in MeanDC become equivalent. As a result, both algorithms have the same asymptotic time complexity of  $O(\log p \cdot |E|^{1.5})$ . Hence, MeanDC is never weaker than BinaryDC.

Beyond these theoretical considerations, experimental results further demonstrate that MeanDC achieves substantially lower running time than BinaryDC.

## 5 NOVEL CORE-PARTITION TECHNIQUE

Although MeanDC can divide the graph more rapidly, it still performs maximum-flow computations on the entire graph at recursion depth 1, which incurs substantial overhead. To address this limitation, we leverage the Sandwich Theorem (Theorem 2), which relates density decomposition to core decomposition, and propose a novel *core-partition* technique. This technique first partitions the graph into approximately  $\log_2 \delta$  subgraphs using core decomposition before executing any maximum-flow computations, thereby making the subsequent divide-and-conquer process significantly more localized and efficient.

**The core-partition technique.** By the Sandwich Theorem, we have  $C_{2k} \subseteq D_k \subseteq C_k$ . Therefore, if both  $C_k$  and  $C_{2k}$  are available,  $D_k$  can be computed locally by invoking LocalGetDk( $G, k, C_k, C_{2k}$ ), which restricts the computation to  $C_k \setminus C_{2k}$  and runs in  $O(|E(C_k \setminus C_{2k})|^{1.5})$  time. Using this relationship, we partition the graph into a sequence of disjoint regions:  $C_1 \setminus C_2, C_2 \setminus C_4, C_4 \setminus C_8, \dots$ , until reaching  $C_{2^i} \setminus C_{2^{i+1}} = \emptyset$ . Within each part  $C_{2^i} \setminus C_{2^{i+1}}$ , we perform LocalGetDk( $G, 2^i, C_{2^i}, C_{2^{i+1}}$ ) to obtain  $D_{2^i}$ . Let  $\mathcal{D}_C$  be the set of computed layers  $D_{2^i}$ . We define the *gap function* as the index intervals of the remaining layers to be computed, which serve as inputs to the subsequent divide-and-conquer procedure.

**DEFINITION 4. (Gap function)** *Given a set of already computed layers  $\widetilde{\mathcal{D}} = \{D_{k_1}, D_{k_2}, \dots, D_{k_t}\}$  with  $k_1 < k_2 < \dots < k_t$ , we define  $\text{Gap}(\widetilde{\mathcal{D}}) = \{(k_i, k_{i+1}) \mid k_{i+1} - k_i > 1, i = 1, 2, \dots, t - 1\}$ .*

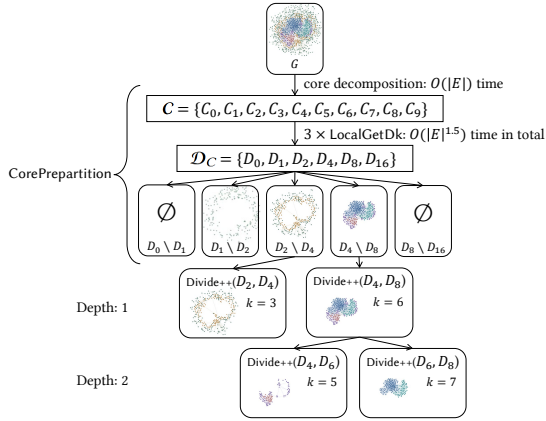


Figure 3: Overview of the CoreDC workflow on Wiki-Vote.

For example, suppose that core-prepartition computes layers  $\mathcal{D}_C = \{D_0, D_1, D_2, D_4, D_8\}$ . Then, the index ranges of layers that remain to be computed are  $\text{Gap}(\mathcal{D}_C) = \{(2, 4), (4, 8)\}$ .

**The novel CoreDC algorithm.** Using the core-prepartition technique, we design the algorithm CoreDC, as shown in Algorithm 5. The algorithm first enters the procedure CorePreparation, which computes the core decomposition  $\mathcal{C} = \{C_0, C_1, \dots, C_\delta\}$  (line 7). Then, according to the Sandwich Theorem, the algorithm directly obtains three layers  $D_0, D_1$ , and  $D_{2^{\lfloor \log_2 \delta \rfloor + 1}}$  (line 8). Next, the algorithm performs the prepartitioning of the graph: for each region  $C_{2^i} \setminus C_{2^{i+1}}$ , it computes  $D_{2^i}$  (lines 9–11). The enumeration of  $i$  terminates when  $i = \lfloor \log_2 \delta \rfloor$ , at which point  $2^{i+1} > \delta$  and thus  $C_{2^{i+1}} = \emptyset$ , indicating that the graph has been completely partitioned. All layers  $\mathcal{D}_C$  computed by core-prepartition are then returned to CoreDC. The algorithm subsequently computes the remaining layers by invoking algorithm  $\text{Divide++}(D_l, D_u)$  for each interval  $(l, u) \in \text{Gap}(\mathcal{D}_C)$  (lines 3–4), and finally returns the full density decomposition  $\mathcal{D}$  (line 5). Next, we present a running example of CoreDC, followed by proofs of its correctness and complexity.

**EXAMPLE 4.** The execution of CoreDC on the example graph in Figure 1 is illustrated in Figure 3. The algorithm first computes the core decomposition  $\mathcal{C}$  and obtains  $\delta = 9$ . It then directly derives three layers  $D_0 \leftarrow C_0, D_1 \leftarrow C_1$ , and  $D_{16} \leftarrow \emptyset$ . Next, the algorithm invokes LocalGetDk on the three subgraphs  $C_2 \setminus C_4, C_4 \setminus C_8$ , and  $C_8 \setminus C_{16}$  to locally obtain  $D_2, D_4$ , and  $D_8$ , respectively. In total, core prepartition computes six layers of density decomposition,  $\mathcal{D}_C = \{D_0, D_1, D_2, D_4, D_8, D_{16}\}$ , which partition the graph into five disjoint parts:  $D_0 \setminus D_1, D_1 \setminus D_2, D_2 \setminus D_4, D_4 \setminus D_8$ , and  $D_8 \setminus D_{16}$ . The algorithm then computes  $\text{Gap}(\mathcal{D}_C) = \{(2, 4), (4, 8), (8, 16)\}$  and subsequently invokes Divide++ to compute the remaining layers.

**THEOREM 4.** The procedure CorePreparation has a time complexity of  $O(|E|^{1.5})$ . The CoreDC algorithm correctly returns the density decomposition  $\mathcal{D}$ , with a time complexity of  $O(\log p \cdot \sum_{(l,u) \in \text{Gap}(\mathcal{D}_C)} |E(D_l \setminus D_u)|^{1.5})$  and a space complexity of  $O(|E|)$ .

**PROOF.** For correctness, first, by setting  $k = 0$  and  $k = 1$  in the Sandwich Theorem, we obtain  $D_0 = C_0$  and  $D_1 = C_1$ . Since  $\delta \geq p$ , it follows that  $2^{\lfloor \log_2 \delta \rfloor + 1} > p$  and thus  $D_{2^{\lfloor \log_2 \delta \rfloor + 1}} = \emptyset$ . Therefore, the three layers  $D_0, D_1$ , and  $D_{2^{\lfloor \log_2 \delta \rfloor + 1}} = \emptyset$  obtained in line 8 are correct. Then, according to the Sandwich Theorem, we have  $C_{2^{i+1}} \subseteq D_{2^i} \subseteq C_{2^i}$ , which ensures that the algorithm can correctly compute  $D_{2^i}$  via LocalGetDk. Afterwards, the algorithm invokes Divide++

to compute all remaining layers. Since the termination condition guarantees that  $D_{2^{\lfloor \log_2 \delta \rfloor + 1}} = \emptyset$ , all remaining layers are covered and computed, ensuring that the algorithm correctly returns the complete density decomposition.

For time complexity, we first analyze the CorePreparation procedure. The core decomposition in line 7 can be computed in  $O(|E|)$  time [4]. In lines 9–11, the time complexity of all LocalGetDk calls is  $O(\sum_{i=1}^{\lfloor \log_2 \delta \rfloor} |E(C_{2^i} \setminus C_{2^{i+1}})|^{1.5}) \leq O(|E|^{1.5})$ . Therefore, the overall time complexity of CorePreparation is  $O(|E|^{1.5})$ . In addition to this cost, the algorithm CoreDC incurs the divide-and-conquer overhead in lines 3–4. These divide-and-conquer calls generate  $|\text{Gap}(\mathcal{D}_C)|$  recursion trees, whose total cost at recursion depth 1 is  $T_1 = O(\sum_{(l,u) \in \text{Gap}(\mathcal{D}_C)} |E(D_l \setminus D_u)|^{1.5})$ . According to the proof of Lemma 1, the total cost of recursive calls at each subsequent depth is bounded by  $T_1$ , and the recursion depth is bounded by  $O(\log p)$ . Hence, the overall time complexity is  $O(\log p \cdot \sum_{(l,u) \in \text{Gap}(\mathcal{D}_C)} |E(D_l \setminus D_u)|^{1.5})$ . For space complexity, since LocalGetDk requires  $O(|E|)$  space, the total space complexity of the algorithm is also  $O(|E|)$ .  $\square$

**Discussion: why core-prepartition is powerful.** In the divide-and-conquer processing, the shallow recursion levels typically dominate the overall running time, since the subgraphs at these levels remain large. When the input graph is extremely large (e.g., billions of edges), each network flow computation becomes prohibitively expensive. Therefore, it is crucial to partition the graph into smaller pieces as early as possible before performing network flow computations. From this perspective, the differences among the three algorithms are substantial. In BinaryDC, each partition requires  $O(\log p)$  maximum-flow computations, resulting in significant overhead. MeanDC improves upon this design by reducing the cost to a single maximum-flow computation per partition. In contrast, CoreDC leverages the core-prepartition technique to achieve much faster initial partitioning. Specifically, while MeanDC spends  $O(|E|^{1.5})$  time at recursion depth 1 to divide the graph into two subgraphs, the CorePreparation procedure spends the same  $O(|E|^{1.5})$  time to partition the graph into approximately  $\log_2 \delta$  subgraphs. This rapid initial partitioning substantially reduces the size of subgraphs processed in subsequent recursion levels, leading to significantly improved efficiency in practice and highlighting the effectiveness of the core-prepartition technique.

## 6 THE PROPOSED PARALLEL ALGORITHMS

In this section, we study parallel algorithms for computing density decomposition. First, we parallelize the four sequential algorithms introduced in the previous sections. Then, we present a novel, parallel-friendly *heat-extraction* technique, which simulates heat diffusion to iteratively approximate IDN values and extract a subset of layers. Based on this technique, we develop the algorithm HeatDC. Finally, by combining core-prepartition with the heat-extraction technique, we propose our most efficient parallel algorithm, CoreHeatDC.

**Parallel computation model.** We adopt the classical *work-span model* to analyze the parallel complexity of our algorithms. In this model, all threads share a single global memory space and can concurrently access the same data. The *work*  $W$  denotes the total amount of computation performed across all threads, while the *span*  $S$  represents the length of the longest chain of dependent operations, which serves as the theoretical lower bound of the parallel running time. Under a randomized work-stealing scheduler [3], the expected running time on  $P$  processors is  $O(W/P + S)$ . In addition to work

and span, we also measure the *peak memory usage* to evaluate the space efficiency of our parallel algorithms. We employ a thread-pool model to efficiently manage threads. A fixed number of worker threads is created at initialization, and a global task queue stores all computational tasks generated during execution. Each idle thread in the pool continuously fetches available tasks from the queue and executes them until all tasks are completed. This design ensures stable and efficient utilization of computational resources.

It is worth noting that max-flow computation is notoriously difficult to parallelize, and the parallelization of maximum-flow algorithms constitutes a separate research area [31, 47, 49]. In this work, we focus on parallelizing the density decomposition framework itself and treat the maximum-flow computation as a black box. In practice, this component can be readily replaced by a parallel maximum-flow implementation.

## 6.1 Basic Parallel Algorithms

**Parallelized IncrFlow algorithm.** For each increasing value of  $k$ , the computation of  $D_k$  via GetDk is encapsulated as an independent task and submitted to the task queue. The process continues until one of the tasks returns an empty  $D_k$ , at which point no new tasks are generated, and the density decomposition is returned. The parallelized IncrFlow has a work complexity of  $O((p + n_T) \cdot |E|^{1.5})$  and a span of  $O(|E|^{1.5})$ , where  $n_T$  denotes the number of threads. Since each thread constructs its own flow network instance, the peak memory usage is  $O(n_T \cdot |E|)$ , which becomes prohibitive when both the graph size and the number of threads are large.

**Parallelized BinaryDC algorithm.** The key idea in parallelizing BinaryDC exploits its divide-and-conquer structure. When the Divide procedure invokes its two child subprocedures, these two calls are encapsulated as two separate tasks and pushed into the task queue. Because the graph is partitioned into disjoint subgraphs, these subprocedures incur no data races or conflicts. The parallelized BinaryDC algorithm achieves a work complexity of  $O(\log p \cdot |E|^{1.5})$  and a span of  $O(\log p \cdot |E|^{1.5})$ . Since each invocation of LocalGetDk only requires local memory of  $O(|E(D_l \setminus D_u)|)$  and different threads process disjoint regions, the peak memory usage remains bounded by  $O(|E|)$ .

**Parallelized MeanDC algorithm.** The parallelization strategy for MeanDC follows that of BinaryDC. In Divide++, two recursive calls are encapsulated as independent tasks and submitted to the task queue. Consequently, the parallelized MeanDC algorithm has a work of  $O(\log p \cdot |E|^{1.5})$ , a span of  $O(\log p \cdot |E|^{1.5})$ , and a peak memory of  $O(|E|)$ , matching those of parallelized BinaryDC.

**Parallelized CoreDC algorithm.** In CorePrepartition, the processing of the  $\lfloor \log_2 \delta \rfloor$  values of  $i$  in lines 9–11 are independent and can be executed in parallel. By summing over all  $i$ , the work of CorePrepartition is  $W_{CP} = O(\sum_{i=1}^{\lfloor \log_2 \delta \rfloor} |E(C_{2^i} \setminus C_{2^{i+1}})|^{1.5})$ , the span is  $S_{CP} = O(\max_{i=1}^{\lfloor \log_2 \delta \rfloor} |E(C_{2^i} \setminus C_{2^{i+1}})|^{1.5})$ , and the peak memory usage is  $O(|E|)$ . The subsequent divide-and-conquer phase in lines 3–4 is parallelized in the same manner as in MeanDC. Consequently, CoreDC has a work of  $O(W_{CP} + \log p \cdot \sum_{(l,u) \in \text{Gap}(\mathcal{D}_C)} |E(D_l \setminus D_u)|^{1.5})$ , a span of  $O(S_{CP} + \log p \cdot \max_{(l,u) \in \text{Gap}(\mathcal{D}_C)} |E(D_l \setminus D_u)|^{1.5})$ , and a peak memory usage of  $O(|E|)$ .

**Discussion.** Among the basic parallel algorithms, the parallelized IncrFlow suffers from excessive memory consumption. In our experiments, on the massive graph SKAll with 1.81B edges and 32 threads, IncrFlow exceeds the 256 GB memory limit. In contrast, the remaining three divide-and-conquer-based algorithms rely on

---

### Algorithm 6: HeatDC( $G$ )

---

**Input:** A graph  $G = (V, E)$ .  
**Output:** The density decomposition  $\mathcal{D}$  of  $G$ .

```

1  $\mathcal{D}_H \leftarrow \text{HeatExtraction}(G)$ ;
2  $\mathcal{D} \leftarrow \mathcal{D}_H$ ;
3 for  $(l, u) \in \text{Gap}(\mathcal{D}_H)$  do // Compute remaining layers
4    $\text{Divide++}(D_l, D_u)$ ;
5 return  $\mathcal{D}$ ;

6 Procedure HeatExtraction( $G$ )
7   Initialize the heat array  $h^0(u) \leftarrow 0$  for each vertex  $u$ ;
8   for  $t = 1, 2, \dots, T$  do // Heat-diffusion iteration
9     foreach  $u \in V$  in parallel do  $h^t(u) \leftarrow h^{t-1}(u)$ ;
10    foreach  $(u, v) \in E$  in parallel do
11      if  $h^{t-1}(u) < h^{t-1}(v)$  then  $h^t(u) \leftarrow h^t(u) + 1$ ;
12      else  $h^t(v) \leftarrow h^t(v) + 1$ ;
13    $\mathcal{D}_H \leftarrow \text{Extract}(G, h^T)$ ; // Algorithm 7
14   return  $\mathcal{D}_H$ ;
```

---

graph partitioning to enable parallelism. As discussed in Section 5, BinaryDC has the slowest graph partitioning process, MeanDC performs better, while CoreDC achieves the fastest partitioning speed. The core-prepartition technique in CoreDC directly splits the graph into approximately  $\lfloor \log_2 \delta \rfloor$  subgraphs, enabling  $\lfloor \log_2 \delta \rfloor$  threads to begin processing immediately. Consequently, in practice, the parallel speedup follows the order  $\text{CoreDC} > \text{MeanDC} > \text{BinaryDC}$ . However, even with this improvement,  $\lfloor \log_2 \delta \rfloor$  is still relatively small. In real-world large graphs, the degeneracy  $\delta$  is typically less than  $2^{14}$  [36], which means that many threads remain idle. As a result, the performance of CoreDC tends to saturate when the number of threads reaches 8 in our experiments.

## 6.2 Novel Parallel Algorithms

To improve parallel scalability by enabling early-stage graph partitioning without network flow computations, we propose a *heat-extraction* technique to extract a subset of layers in the density decomposition. This technique first performs a *heat-diffusion* iterative process (lines 7–12 of Algorithm 6), during which the heat values of vertices converge to their IDNs. After sufficiently accurate IDN approximations are obtained, the technique extracts layers based on the resulting heat values (Algorithm 7). By incorporating the heat-extraction technique, we develop the algorithms HeatDC and CoreHeatDC for computing the density decomposition. We now introduce the heat-extraction technique in detail.

**Heat-diffusion iteration.** To build intuition, consider a surface containing numerous heat-producing units: regions with denser units generate more heat, and heat naturally flows from hotter areas to cooler ones. Over time, the resulting heat distribution reflects the underlying density of these units. Our iterative heat-diffusion process follows exactly the same principle. Here, edges act as heat-producing units, and heat is transferred toward cooler vertices. Vertices in dense regions (i.e., those with higher IDN values) accumulate heat more quickly, whereas vertices in sparser regions accumulate heat more slowly. As the iterations proceed, the heat distribution gradually separates the vertices according to their density levels.

Formally, the pseudocode of the heat-diffusion iteration is outlined in lines 7–12 of Algorithm 6. We initialize each vertex with zero heat. Then, for  $T$  iterations, each vertex copies its previous heat value (line 3). For every edge  $(u, v)$ , the endpoint with the lower heat value receives one unit of heat (lines 5–6). As the iterations progress, vertices continue accumulating heat. As we will show, when the number of iterations  $T \rightarrow \infty$ , it holds that  $\lceil h^T(u)/T \rceil \rightarrow \bar{r}(u)$ ,

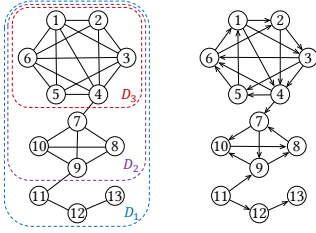


Figure 4: A small example graph and its orientation.

meaning that the heat values naturally separate vertices according to their IDN levels. We next prove this convergence.

**THEOREM 5.** *For the heat-diffusion iteration in lines 7–12 of Algorithm 6, as  $T \rightarrow \infty$ , for every vertex  $u \in V$ ,  $h^T(u)/T$  converges to a limit  $x^*(u)$  satisfying  $\lceil x^*(u) \rceil = \bar{r}(u)$ . The convergence rate is guaranteed by  $\|h^T/T - x^*\|_2 \leq \sqrt{\frac{2(1+\ln T)d_{\max}|E|}{T}}$ , where  $d_{\max} = \max_{u \in V} d(u)$  denotes the maximum degree.*

**PROOF.** The detailed proof is presented in the appendix.  $\square$

In practice, the heat-diffusion iteration efficiently and in a parallel-friendly manner produces high-quality approximations of IDN values within only a few hundred iterations. In our implementation, during each iteration, we evenly partition the edge set into  $128 \times n_t$  chunks, where  $n_t$  is the number of threads. For each chunk, the update  $h^t$  task in lines 10–12 is pushed into the thread pool, creating a total of  $128 \times n_t$  tasks. This design achieves sufficiently fine-grained parallelism to balance the workload across threads, while avoiding excessive scheduling overhead. Consequently, each iteration proceeds with high parallel efficiency.

**The heat-extraction technique.** Given the heat-diffusion result  $h^T$ , we now consider how to extract layers from  $h^T$  without max-flow. We adopt an *orientation*-based construction and verification approach. Given an undirected graph  $G = (V, E)$ , an *orientation*  $\vec{G} = (V, \vec{E})$  is obtained by assigning a direction to each undirected edge. Let  $\vec{d}(u)$  denote the indegree of vertex  $u$  in  $\vec{G}$ . A *path*  $s \rightsquigarrow t$  in an orientation is a sequence of vertices  $s = u_0, u_1, \dots, u_\ell = t$  such that  $(u_{i-1}, u_i) \in \vec{E}$  for all  $i = 1, \dots, \ell$ . The following lemma illustrates how to extract a layer  $D_k$  from an orientation.

**LEMMA 2.** [56, 60] *Given a non-negative integer  $k$ , if an orientation  $\vec{G}$  contains no path  $s \rightsquigarrow t$  such that  $\vec{d}(s) < k - 1 < \vec{d}(t)$ , then  $D_k = \{u \mid \vec{d}(u) > k - 1 \text{ or } \exists u \rightsquigarrow v, \vec{d}(v) > k - 1\}$ .*

By Lemma 2, once an orientation satisfying the no-path condition is available, the layer  $D_k$  can be computed in linear time  $O(|E|)$  via a single breadth-first search (BFS) initiated from all vertices with indegree greater than  $k - 1$ .

**EXAMPLE 5.** Figure 4 shows an example graph together with one of its orientations, illustrating the layers  $D_1$ ,  $D_2$ , and  $D_3$ . For  $k = 3$ , the orientation satisfies the no-path condition, and thus  $D_3$  consists of all vertices with indegree greater than 2, namely  $\{4, 6\}$ , along with the vertices that can reach them, namely  $\{1, 2, 3, 5\}$ . For  $k = 2$ , there exists a path  $11 \rightsquigarrow 9$  such that  $\vec{d}(9) > 1 > \vec{d}(11)$ , violating the no-path condition.

Next, we propose a heuristic method, called Extract (Algorithm 7), for constructing an orientation that often satisfies the no-path condition for many values of  $k$ , thereby enabling the extraction of multiple layers. The algorithm takes the heat array  $h$

#### Algorithm 7: Extract( $G, h$ )

---

**Input:** A graph  $G = (V, E)$  and heat array  $h$ .  
**Output:** Extracted layers  $\mathcal{D}_H$ .

- 1 Initialize an orientation  $\vec{G} = (V, \vec{E} = \emptyset)$  and its indegree array  $\vec{d}(\cdot) \leftarrow 0$ ;
- 2 **foreach**  $(u, v) \in E$  **in parallel do** // **Initialize orientation**
- 3   **if**  $h(u) < h(v)$  **then**  $\vec{E} \leftarrow \vec{E} \cup \{(u, v)\}$ ,  $\vec{d}(u) \leftarrow \vec{d}(u) + 1$ ;
- 4   **else**  $\vec{E} \leftarrow \vec{E} \cup \{(v, u)\}$ ,  $\vec{d}(v) \leftarrow \vec{d}(v) + 1$ ;
- 5 **for**  $t = 1, 2, \dots, 10$  **do** // **Balance orientation**
- 6   **foreach**  $(u, v) \in \vec{E}$  **in parallel do**
- 7     **if**  $\vec{d}(v) > \vec{d}(u)$  **then**
- 8       reverse  $(u, v)$  to  $(v, u)$ ,  $\vec{d}(v) \leftarrow \vec{d}(v) - 1$ ,  $\vec{d}(u) \leftarrow \vec{d}(u) + 1$ ;
- 9  $\vec{d}_{\max} = \max_{u \in V} \vec{d}(u)$ ,  $\mathcal{D}_H \leftarrow \{D_0 = V, D_{\vec{d}_{\max}+1} = \emptyset\}$ ;
- 10 **for**  $k = 1, \dots, \vec{d}_{\max}$  **in parallel do** // **Verify the no-path condition**
- 11   **if** there is no path  $s \rightsquigarrow t$ ,  $\vec{d}(s) < k - 1 < \vec{d}(t)$  **then**
- 12      $D_k \leftarrow \{u \mid \vec{d}(u) > k - 1 \text{ or } \exists u \rightsquigarrow v, \vec{d}(v) > k - 1\}$ ;
- 13      $\mathcal{D}_H \leftarrow \mathcal{D}_H \cup \{D_k\}$ ;
- 14 **return**  $\mathcal{D}_H$ ;

---

as input and extracts a subset of layers  $\mathcal{D}_H$ . First, it initializes an empty orientation (line 1). Intuitively, the no-path condition requires that vertices with lower indegree should not point to vertices with higher indegree; equivalently, edges should be oriented from denser regions to sparser ones. Accordingly, during initialization, for each edge  $(u, v)$ , the algorithm orients it toward the vertex with the smaller heat value, corresponding to a sparser region (lines 2–4). The resulting orientation may be noisy, as local fluctuations of  $h$  can lead to imbalances that violate the no-path condition. To mitigate this issue, the algorithm applies a simple yet effective orientation-balancing routine adapted from [58] to smooth the orientation (lines 5–8). The algorithm then proceeds to extract layers. It first initializes two boundary layers,  $D_0 = V$  and  $D_{\vec{d}_{\max}+1} = \emptyset$  (line 9). Subsequently, for each  $k$ , the algorithm checks whether the no-path condition in Lemma 2 is satisfied; if so, it extracts the layer  $D_k$ , adds it to  $\mathcal{D}_H$ , and returns (lines 10–14).

**The HeatDC algorithm.** Building on the heat-extraction technique, we propose HeatDC for computing the density decomposition (Algorithm 6). After executing the procedure HeatExtraction, a subset of layers  $\mathcal{D}_H$  is obtained, while the layers indexed by  $\text{Gap}(\mathcal{D}_H)$  remain uncomputed. The algorithm then invokes Divide++ on each interval in  $\text{Gap}(\mathcal{D}_H)$  to compute the missing layers and finally output the complete density decomposition. We now analyze the complexity of HeatExtraction and HeatDC.

**THEOREM 6.** *The procedure HeatExtraction has work complexity  $W_{HE} = O(T \cdot |E|)$  and span complexity  $S_{HE} = O(T + |E|)$ . HeatDC has work complexity  $O(W_{HE} + \sum_{(l,u) \in \text{Gap}(\mathcal{D}_H)} |E(D_l \setminus D_u)|^{1.5})$ , span complexity  $O(S_{HE} + \max_{(l,u) \in \text{Gap}(\mathcal{D}_H)} |E(D_l \setminus D_u)|^{1.5})$ , and peak memory usage bounded by  $O(|E|)$ .*

**PROOF.** We first analyze the complexity of HeatExtraction. It performs  $T$  rounds of heat-diffusion iterations, each scanning all edges once, resulting in  $O(T \cdot |E|)$  total work. Since these rounds are sequentially dependent, they contribute a span of  $O(T)$ . After the iterations, HeatExtraction executes the Extract procedure once, which has both work and span  $O(|E|)$ , as it requires only a constant number of load-balancing rounds and then verifies the no-path conditions for all  $k$  by scanning vertices without revisiting. Therefore, the work complexity of HeatExtraction is  $W_{HE} = O(T \cdot |E|)$  and its span complexity is  $S_{HE} = O(T + |E|)$ . After HeatExtraction completes, HeatDC invokes Divide++ to compute the remaining

**Algorithm 8:** CoreHeatDC( $G$ )

---

**Input:** A graph  $G = (V, E)$ .  
**Output:** The density decomposition  $\mathcal{D}$  of  $G$ .  
1 Run  $\mathcal{D}_C \leftarrow \text{CorePrepartition}(G)$  and  $\mathcal{D}_H \leftarrow \text{HeatExtraction}(G)$  in parallel;  
2  $\mathcal{D} \leftarrow \mathcal{D}_C \cup \mathcal{D}_H$ ;  
3 **for**  $(l, u) \in \text{Gap}(\mathcal{D}_C \cup \mathcal{D}_H)$  **do**   // Compute remaining layers  
4    $\text{Divide++}(D_l, D_u)$ ;  
5 **return**  $\mathcal{D}$ ;

---

layers indexed by  $\text{Gap}(\mathcal{D}_H)$ . Summing over all gaps, HeatDC has work complexity  $O(W_{HE} + \sum_{(l,u) \in \text{Gap}(\mathcal{D}_H)} |E(D_l \setminus D_u)|^{1.5})$ , span complexity  $O(S_{HE} + \max_{(l,u) \in \text{Gap}(\mathcal{D}_H)} |E(D_l \setminus D_u)|^{1.5})$ , and the peak memory  $O(|E|)$ .  $\square$

**The CoreHeatDC algorithm.** The core-prepartition and the heat-extraction techniques share the common goal of precomputing a set of layers to partition the graph. Consequently, the two techniques can be naturally combined by taking the union of the layers precomputed by both methods and applying divide-and-conquer to complete the remaining layers. Based on this idea, we propose the CoreHeatDC algorithm, as shown in Algorithm 8.

The workflow of CoreHeatDC is as follows. First, the algorithm runs the procedures CorePrepartition and HeatExtraction in parallel (line 1). In practical implementations, the tasks generated by these two procedures can be submitted to a global task queue, allowing the thread pool to automatically schedule idle threads to execute them. Next, CoreHeatDC takes the union of the layers obtained from the two techniques (line 2) and computes the remaining layers using Divide++ (lines 3–5). The complexity of CoreHeatDC follows directly from the complexities of CorePrepartition, HeatExtraction, and Divide++, as stated in the following theorem.

**THEOREM 7.** CoreHeatDC has work complexity  $O(W_{CP} + W_{HE} + \sum_{(l,u) \in \text{Gap}(\mathcal{D}_C \cup \mathcal{D}_H)} |E(D_l \setminus D_u)|^{1.5})$ , span complexity  $O(\max\{S_{CP}, S_{HE}\} + \max_{(l,u) \in \text{Gap}(\mathcal{D}_C \cup \mathcal{D}_H)} |E(D_l \setminus D_u)|^{1.5})$ , and peak memory usage bounded by  $O(|E|)$ , where  $W_{CP} = O(\sum_{i=1}^{\lceil \log_2 \delta \rceil} |E(C_{2^i} \setminus C_{2^{i+1}})|^{1.5})$  and  $S_{CP} = O(\max_{i=1}^{\lceil \log_2 \delta \rceil} |E(C_{2^i} \setminus C_{2^{i+1}})|^{1.5})$  are the work and span of CorePrepartition, and  $W_{HE} = O(T \cdot |E|)$  and  $S_{HE} = O(T + |E|)$  are the work and span of HeatExtraction.

### Discussion: combining core-prepartition and heat-extraction.

The combined strategy of core-prepartition and heat-extraction offers several important advantages. (1) Recall that the parallelized core-prepartition procedure can utilize only about  $\log_2 \delta$  threads, leaving the remaining threads idle. These idle threads can instead be used to perform heat-extraction, thereby significantly improving the utilization of parallel resources. (2) The partitions induced by core-prepartition are predictable and relatively balanced, with each layer  $D_{2^i}$  naturally defining one partition. In contrast, heat-extraction relies on a heuristic procedure that may produce uneven layer partitions. In such cases, the more structured partitions provided by core-prepartition help prevent the formation of overly large subgraphs (see, e.g., Exp-5). (3) Applying both techniques simultaneously further refines the graph partitioning before the divide-and-conquer stage, which directly reduces the cost of the subsequent recursive computations. Overall, core-prepartition and heat-extraction are highly complementary. Together, they form a powerful preprocessing strategy for handling large-scale graphs.

## 6.3 Complexity Comparison

Table 1 summarizes the work complexity, span complexity, and peak memory usage of all parallel algorithms. The parallelized IncrFlow

incurs prohibitive peak memory  $O(n_t \cdot |E|)$ , and is thus impractical for massive graphs with many threads. For BinaryDC and MeanDC, although they differ in pivot-selection strategies, both algorithms initiate max-flow computations on the entire graph. As a result, their work and span complexities are inevitably dominated by full-graph max-flow, i.e.,  $O(\log p \cdot |E|^{1.5})$ .

CoreDC introduces an additional core-prepartition phase with work and span  $W_{CP}$  and  $S_{CP}$ , but substantially reduces the workload of the subsequent divide-and-conquer stage by shrinking flow networks from the whole graph to much smaller subgraphs  $D_l \setminus D_u$ . Since the max-flow cost is superlinear in the number of edges (with exponent 1.5), splitting a large instance into multiple smaller gaps not only reduces the total work, but also lowers the span to be governed by the largest gap, i.e.,  $\max_{(l,u) \in \text{Gap}(\mathcal{D}_C)} |E(D_l \setminus D_u)|^{1.5}$ .

HeatDC and CoreHeatDC further improve parallel efficiency by leveraging heat-extraction to heuristically extract many layers and thereby split gaps more finely. Our experiments (Exp-4 and Exp-5) show that heat-extraction can directly extract a substantial fraction of layers (up to 89% across our datasets) and significantly increases the number of gaps (up to 62), which in turn reduces both the runtime and the number of edges involved in subsequent max-flow calls. Moreover, CoreHeatDC combines core prepartitioning and heat extraction to produce the most fine-grained and localized gap partition, enabling the highest degree of parallelism in our framework. Although it introduces extra overhead from both procedures, CoreHeatDC is the most efficient algorithm in practice, achieving the best overall work and span among all parallel approaches.

## 7 EXPERIMENTS

### 7.1 Experimental Setup

**Algorithms.** We implemented and parallelized the existing algorithms for density decomposition, including IncrFlow [56, 60] and BinaryDC (Algorithm 3) [56, 60] as baselines. We also implemented our proposed algorithms MeanDC (Algorithm 4), CoreDC (Algorithm 5), HeatDC (Algorithm 6), and CoreHeatDC (Algorithm 8). Unless otherwise specified, the number of iterations  $T$  used in the HeatExtraction procedure of HeatDC and CoreHeatDC is set to 500. All experiments were conducted on a Ubuntu PC equipped with a 2.7 GHz AMD Ryzen Threadripper PRO 3995WX CPU and 256 GB of main memory. All programs were compiled with g++ using the -O3 optimization flag.

**Datasets.** We evaluated our algorithms on a variety of real-world graphs from different domains, with details summarized in Table 2. LiveJ [23] represents friendship relationships among users in the LiveJournal online community. HW09 [23] captures co-acting relationships between actors in movies from IMDB. DBpedia [34] and WikiEn [41] corresponds to wikilink relationships between pages in Wikipedia. P2P [41] represents the eDonkey peer-to-peer network. Twitter [41] captures follow relationships among Twitter users. CC12 [41], ITAll [41], GSH [23], and SKAll [41] are web networks representing hyperlink relationships. All directed graphs were converted to undirected graphs for evaluation.

### 7.2 Experimental Results

**Exp-1: Runtime comparison under the sequential setting.** Using a single thread, the running times of all algorithms are shown in Figure 5. As expected, our proposed algorithms MeanDC and CoreDC consistently outperform the baseline methods IncrFlow and BinaryDC. Compared with BinaryDC, MeanDC achieves higher efficiency by eliminating the binary search for the pivot. For instance, on SKAll, BinaryDC and MeanDC take 9,595 seconds

**Table 1: Complexity of different parallel algorithms.**

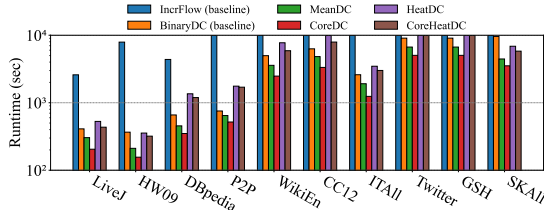
$n_t$  is the number of threads.  $W_{CP} = O(\sum_{i=1}^{\lceil \log_2 \delta \rceil} |E(C_{2i} \setminus C_{2i+1})|^{1.5})$  and  $S_{CP} = O(\max_{i=1}^{\lceil \log_2 \delta \rceil} |E(C_{2i} \setminus C_{2i+1})|^{1.5})$  are the work and span of CorePrepartition.  $W_{HE} = O(T \cdot |E|)$  and  $S_{HE} = O(T + |E|)$  are the work and span of HeatExtraction.

Algorithm	Work complexity	Span complexity	Peak memory
IncrFlow	$O((p + n_t) \cdot  E ^{1.5})$	$O( E ^{1.5})$	$O(n_t \cdot  E )$
BinaryDC	$O(\log p \cdot  E ^{1.5})$	$O(\log p \cdot  E ^{1.5})$	$O( E )$
MeanDC	$O(\log p \cdot  E ^{1.5})$	$O(\log p \cdot  E ^{1.5})$	$O( E )$
CoreDC	$O(W_{CP} + \log p \cdot \sum_{(l,u) \in \text{Gap}(\mathcal{D}_C)}  E(D_l \setminus D_u) ^{1.5})$	$O(S_{CP} + \log p \cdot \max_{(l,u) \in \text{Gap}(\mathcal{D}_C)}  E(D_l \setminus D_u) ^{1.5})$	$O( E )$
HeatDC	$O(W_{HE} + \log p \cdot \sum_{(l,u) \in \text{Gap}(\mathcal{D}_H)}  E(D_l \setminus D_u) ^{1.5})$	$O(S_{HE} + \log p \cdot \max_{(l,u) \in \text{Gap}(\mathcal{D}_H)}  E(D_l \setminus D_u) ^{1.5})$	$O( E )$
CoreHeatDC	$O(W_{CP} + W_{HE} + \log p \cdot \sum_{(l,u) \in \text{Gap}(\mathcal{D}_C \cup \mathcal{D}_H)}  E(D_l \setminus D_u) ^{1.5})$	$O(\max\{S_{CP}, S_{HE}\} + \log p \cdot \max_{(l,u) \in \text{Gap}(\mathcal{D}_C \cup \mathcal{D}_H)}  E(D_l \setminus D_u) ^{1.5})$	$O( E )$

**Table 2: Datasets statistics**

$p$  is the number of layers in the density decomposition, i.e., the maximum integer such that  $D_p \neq \emptyset$ .

Name	Type	$ V $	$ E $	$p$
LiveJ	Social	5.46M	78.0M	402
HW09	Social	1.14M	113M	2,208
DBpedia	Wikilink	14.0M	130M	689
P2P	Peer-to-peer	5.79M	148M	751
WikiEn	Wikilink	27.2M	543M	581
CC12	Web	42.9M	583M	2,165
ITAll	Web	41.3M	1.03B	2,009
Twitter	Social	41.7M	1.20B	1,644
GSH	Web	68.7M	1.50B	6,094
SKAll	Web	50.6M	1.81B	2,258


**Figure 5: Running time of all algorithms in sequential setting (using one thread).**

and 4,449 seconds, respectively, achieving a 2.2 $\times$  speedup. Furthermore, benefiting from the core-prepartition technique, CoreDC outperforms both BinaryDC and MeanDC on all datasets. For example, on ITAll, the running times of BinaryDC, MeanDC, and CoreDC are 2,599, 1,912, and 1,242 seconds, yielding speedups of 2.1 $\times$  and 1.5 $\times$ , respectively. In contrast, HeatDC and CoreHeatDC, which incorporate heat-extraction technique, are relatively slow in the single-thread setting. This is because heat extraction requires multiple heat-diffusion iterations that repeatedly scan the entire edge set, making it more suitable for the parallel setting. Overall, these results show that CoreDC achieves the best performance in the sequential setting.

**Exp-2: Runtime comparison under the parallel setting.** In this experiment, we vary the number of threads in  $\{1, 2, 4, 8, 16, 24, 32\}$  to evaluate the parallel efficiency of all algorithms. The runtime results are shown in Figure 6, and the self-relative speedups on DBpedia and CC12 are reported in Figure 7 (similar trends are observed on the other datasets). For BinaryDC, MeanDC, and CoreDC, the speedups saturate at approximately 2, 4, and 8 threads, respectively. This is because their parallelization primarily relies on the divide-and-conquer procedure to partition the graph, which produces tasks that are not sufficiently fine-grained. Moreover, as discussed in Section 5, CoreDC partitions the graph faster than MeanDC, while MeanDC in turn improves upon BinaryDC. This leads to increasingly finer-grained tasks and, correspondingly, increasing parallel

speedups from BinaryDC to MeanDC to CoreDC. For instance, on CC12 with 8 threads, the speedup ratios of BinaryDC, MeanDC, and CoreDC are 1.26, 2.72, and 4.73, respectively.

In contrast, IncrFlow, HeatDC, and CoreHeatDC continue to achieve speedups as the number of threads increases on most datasets, indicating a better ability to exploit parallel resources. With 32 threads, their self-relative speedups on DBpedia are 16.4, 9.6, and 17.7, respectively. Compared with HeatDC, CoreHeatDC achieves higher speedups by combining core-prepartition and heat-extraction to further subdivide the graph, resulting in finer-grained tasks during divide-and-conquer. For example, on Twitter, HeatDC and CoreHeatDC take 1,466 and 550 seconds with 32 threads, respectively, yielding a 2.66 $\times$  advantage for CoreHeatDC.

Overall, in the parallel setting, although IncrFlow achieves high speedups, it times out on large graphs and runs out of memory (OOM) on the largest dataset SKAll. For BinaryDC, MeanDC, and CoreDC, they fail to fully exploit parallel resources and thus exhibit limited speedups. When using 32 threads, CoreHeatDC is consistently faster than all other algorithms across all datasets. For example, on SKAll, BinaryDC, MeanDC, CoreDC, HeatDC, and CoreHeatDC take 6340, 1751, 1265, 1153, and 676 seconds, respectively. This implies that CoreHeatDC achieves speedups of 9.4 $\times$ , 2.6 $\times$ , 1.9 $\times$ , and 1.7 $\times$  over other algorithms, making it the best choice in highly parallel settings.

**Exp-3: Comparison of peak memory usage.** In this experiment, we compare the peak memory usage of all algorithms under both sequential and parallel settings. The results are reported in Figure 8. Due to space constraints, we present results only on the medium-sized graph P2P and the large graph SKAll; similar trends are observed on the remaining datasets. As expected, the peak memory usage of IncrFlow increases rapidly with the number of threads, and it even runs out of memory (OOM) on SKAll. This is because each thread in IncrFlow constructs its own flow network, which leads to prohibitively high memory consumption in the multi-threaded setting. In contrast, all other algorithms exhibit low peak memory usage, since their memory usage is bounded by the graph size, i.e.,  $O(|E|)$ . These results confirm that our proposed algorithms are memory-efficient and thus well-suited for large-scale graphs.

**Exp-4: Divide-and-conquer workload comparison.** In this experiment, we compare the workload of the divide-and-conquer procedures across different algorithms at various recursion depths. For a given depth  $h$ , let  $\mathcal{P}_h$  denote the set of parameter pairs  $(D_l, D_u)$  corresponding to the recursive calls of Divide/Divide++ at depth  $h$  in the recursion tree. Let  $f(D_l, D_u)$  denote the runtime of a Divide/Divide++ call with parameters  $(D_l, D_u)$  (excluding the time spent in deeper recursive calls). At each depth  $h$ , we evaluate four metrics: (1) *total runtime*, defined as  $\sum_{(D_l, D_u) \in \mathcal{P}_h} f(D_l, D_u)$ ;

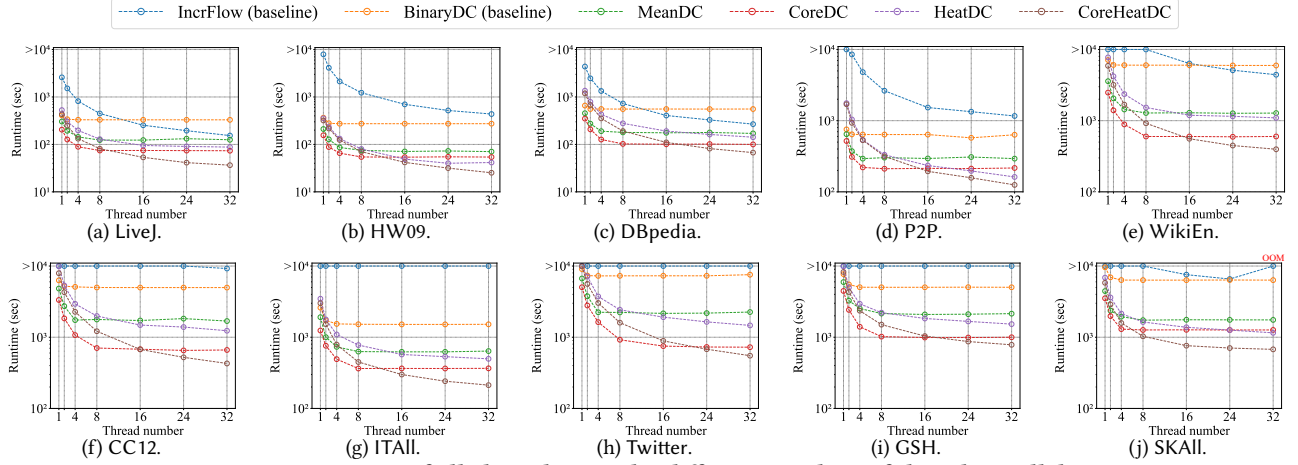
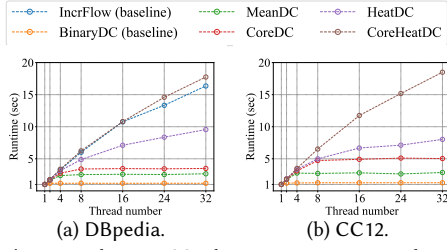


Figure 6: Running time of all algorithms under different numbers of threads on all datasets.



Note: IncrFlow is not shown in CC12 because its runtime exceeds  $10^4$  seconds.

Figure 7: Self-relative speedup on datasets DBpedia and CC12.

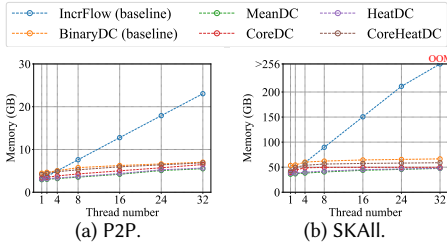


Figure 8: Peak memory usage on datasets P2P and SKAll.

(2) *average runtime*, defined as  $\frac{\sum_{(D_l, D_u) \in \mathcal{P}_h} f(D_l, D_u)}{|\mathcal{P}_h|}$ ; (3) *total edges*, defined as  $\sum_{(D_l, D_u) \in \mathcal{P}_h} |E(D_l, D_u)|$ ; and (4) *average edges*, defined as  $\frac{\sum_{(D_l, D_u) \in \mathcal{P}_h} |E(D_l, D_u)|}{|\mathcal{P}_h|}$ . Figure 9 reports the results on the Twitter dataset; similar trends are observed on the other datasets.

As shown, BinaryDC performs  $O(\log p)$  max-flow computations on the entire graph at depth 1, resulting in an overwhelming runtime cost. In contrast, MeanDC partitions the graph faster, spending significantly less time at shallow depths and quickly progressing to deeper depths. Consequently, the majority of its max-flow computations occur at depths 6–11, where the induced subgraphs are substantially smaller and more localized. CoreDC further improves locality: due to core-partitioning, both the average runtime and the average number of edges at each depth are significantly lower than those of BinaryDC and MeanDC, highlighting the effectiveness of the core-partitioning technique. The heat-extraction technique employed in HeatDC and CoreHeatDC also substantially

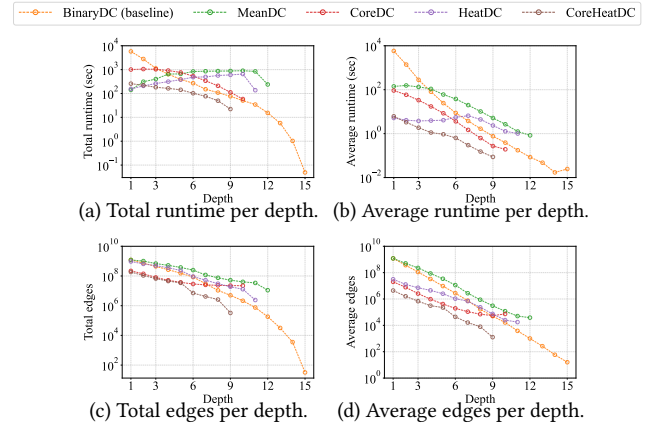


Figure 9: Divide-and-conquer workload by recursion depth on dataset Twitter.

Table 3: Number of layers and gaps of  $\mathcal{D}_C$  and  $\mathcal{D}_H$ .

Dataset	$p$	$ \mathcal{D}_C $	$ \mathcal{D}_H $	$ \text{Gap}(\mathcal{D}_C) $	$ \text{Gap}(\mathcal{D}_H) $	$ \text{Gap}(\mathcal{D}_C \cup \mathcal{D}_H) $
LiveJ	402	10	166	8	17	23
HW09	2,208	13	1,267	11	62	70
DBpedia	689	11	398	9	15	22
P2P	751	11	115	9	63	68
WikiEn	581	11	159	9	11	18
CC12	2,165	13	1,592	11	9	17
ITAll	2,009	12	1,592	10	13	20
Twitter	1,644	12	154	10	31	40
GSH	6,094	14	5,430	12	23	31
SKAll	2,258	13	1,812	11	40	46

enhances locality and efficiency. Among all methods, CoreHeatDC consistently outperforms the others across almost all recursion depths, demonstrating that the combined use of core-partitioning and heat-extraction enables a highly localized and efficient divide-and-conquer process.

**Exp-5: Layers and gaps in core-partitioning and heat-extraction.** In this experiment, we analyze the layers and gaps produced by core-partitioning, heat-extraction, and their combination. Table 3 reports the number of layers and gaps generated by  $\mathcal{D}_C$  and  $\mathcal{D}_H$ . Figures 10 and 11 visualize the positions of extracted

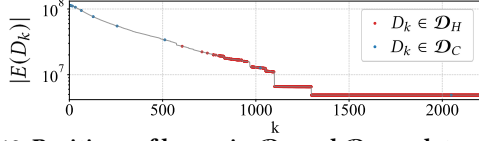
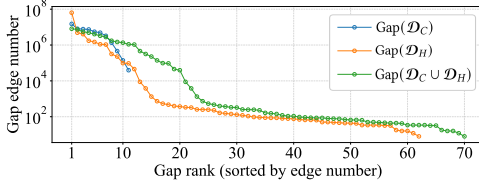


Figure 10: Positions of layers in  $\mathcal{D}_C$  and  $\mathcal{D}_H$  on dataset HW09.



The edge number of a gap  $(l, u)$  is defined as  $|E(D_l \setminus D_u)|$ . The gaps are sorted in descending order of edge numbers for better visualization.

Figure 11: Gap size of core-prepartition and heat-extraction on dataset HW09.

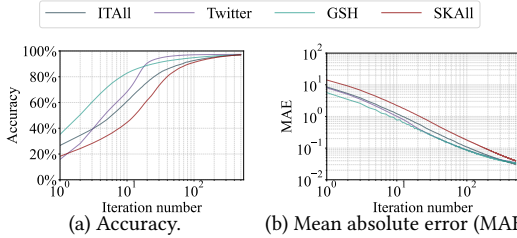


Figure 12: Effect of the number of heat-diffusion iterations on the quality of IDN estimation.

layers and the sizes of resulting gaps on the HW09 dataset; similar results are observed on other datasets.

As shown, the layers  $\mathcal{D}_C$  obtained by core prepartitioning are stable and predictable in number. In contrast, the layers  $\mathcal{D}_H$  extracted by heat extraction exhibit more irregularity: it is able to extract a large number of layers, including many consecutive layers, but may also produce very large gaps. For example, the gap  $(D_2, D_{605})$  in Figure 10 corresponds to the largest gap in Figure 11, with an edge count of 63.6M. Compared with using either technique alone, combining core prepartitioning and heat extraction yields several important advantages. (1) As shown in Table 3 and Figure 11, the combined approach produces a larger number of gaps, each with substantially fewer edges. This allows more gaps to be processed in parallel at the early stage of divide-and-conquer, improving locality and reducing thread idle time. (2) As illustrated in Figure 10, core prepartitioning provides stable partitions, especially for *small*  $k$ , whereas heat extraction tends to extract many layers for *large*  $k$ . The two techniques are therefore highly complementary, effectively avoiding large gaps.

**Exp-6: Effect of the number of heat-diffusion iterations.** In this experiment, we evaluate the impact of the number of heat-diffusion iterations  $T$  on the quality of heat diffusion. We vary  $T$  from 1 to 500 and record the accuracy and the mean absolute error (MAE) at each iteration. The accuracy is defined as the fraction of vertices whose approximated IDN matches the exact IDN, i.e.,  $\frac{| \{u \in V \mid |h^T(u)/T| = f(u) \} |}{|V|}$ . The MAE is defined as the mean absolute error of the IDN approximation, i.e.,  $\frac{\sum_{u \in V} |h^T(u) - f(u)|}{|V|}$ . The results

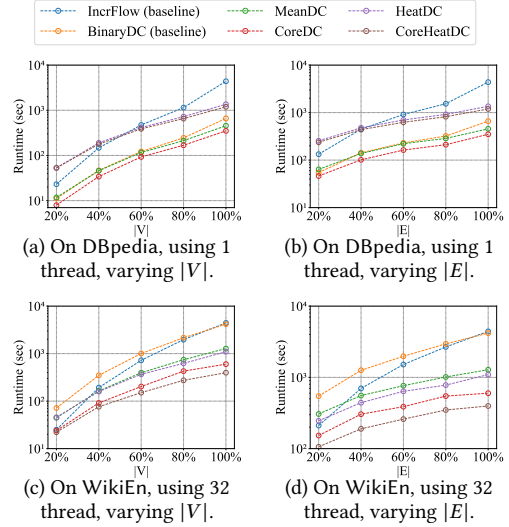


Figure 13: Scalability test on datasets DBpedia and WikiEn.

on ITAll, Twitter, GSH, and SKAll are shown in Figure 12 (similar trends are observed on other datasets). As shown, as  $T$  increases, the accuracy rises rapidly while the MAE decreases sharply. Notably, even with a relatively small number of iterations (e.g.,  $T = 100$ ), the accuracy already reaches 91.0%–97.0% across the four datasets with a relatively low MAE. This demonstrates that heat-diffusion can produce high-quality IDN approximations quickly. When  $T$  reaches 500, the accuracy further improves to 97.0%–97.4%, and the MAE drops to as low as 0.031–0.040, indicating that the vast majority of vertices have converged to their exact IDN. These results confirm that heat diffusion is highly effective for approximating IDN.

**Exp-7: Scalability test.** In this experiment, we evaluate the scalability of different algorithms by varying the graph size. We use DBpedia for the sequential setting and WikiEn for the parallel setting (similar results are observed on other datasets). Specifically, we randomly sample  $\{20\%, 40\%, 60\%, 80\%\}$  of the vertex set  $V$  and the edge set  $E$  from each dataset for testing. The results for the sequential setting are shown in Figures 13a and 13b. As shown, the running time of IncrFlow increases sharply as the graph size grows. The algorithms HeatDC and CoreHeatDC also exhibit relatively high running times, since they rely on the heat-extraction technique, which is more suitable for the parallel setting. In contrast, CoreDC consistently achieves the lowest running time across all graph sizes, demonstrating its high scalability in the sequential setting. The results for the parallel setting are reported in Figures 13c and 13d. As shown, in the highly parallel setting, CoreHeatDC is the fastest algorithm across all graph scales, and its running time grows smoothly with increasing graph size, demonstrating the excellent scalability of CoreHeatDC in the parallel setting.

## 8 RELATED WORKS

**Density-based graph decomposition.** The density of a graph is commonly defined as the ratio between the number of edges and the number of vertices [9, 15, 16, 21, 28]. Graph decomposition models based on this notion of density have attracted increasing attention in recent years, including locally-dense subgraph (LDS) decomposition [12, 18, 21, 50], top- $k$  locally densest subgraphs [38, 40, 51], and density decomposition [10, 11, 18, 56, 57, 59, 60]. The definition of density decomposition was first introduced in [11], together

with a basic path-reversal algorithm with time complexity  $O(|E|^2)$ . Subsequent work [56, 60] incorporated flow network techniques into density decomposition computation and developed divide-and-conquer algorithms, reducing the time complexity to  $O(\log p \cdot |E|^{1.5})$ . These works also established a structural connection between density decomposition and LDS decomposition, showing that the layers of density decomposition form a subset of the LDS decomposition layers. Density decomposition has also been extended to bipartite graphs and directed graphs in [57, 59]. Despite this progress, to the best of our knowledge, no prior work has studied parallel algorithms for density decomposition. Our work fills this gap by designing efficient parallel algorithms and scaling density decomposition to graphs with billions of edges.

**Cohesive subgraph decomposition.** Beyond density-based decompositions, a wide range of cohesive subgraph decompositions have been studied based on alternative structural notions. Representative examples include degree-based core decomposition [4, 26, 27, 37, 43], triangle-based truss decomposition [20, 29, 52], connectivity-based  $k$ -edge-connected subgraph decomposition [13, 14], and clique-based nucleus decomposition [44–46, 48]. Among these methods, core decomposition is often used as an approximation to density-based decompositions. For example, [50] employs core decomposition as a 2-approximation for LDS decomposition; [11] shows that the density of the  $k$ -core is a 2-approximation with respect to  $k$ ; and [56, 60] establish the sandwich relationship  $C_{2k} \subseteq D_k \subseteq C_k$  between density decomposition and core decomposition. However, as observed in [11, 50, 56, 59, 60], compared with classical core decomposition, density decomposition is more sensitive to density rather than merely degree constraints, and can therefore characterize dense regions in graphs more precisely.

## 9 CONCLUSION

In this paper, we study the problem of efficiently computing the density decomposition of large graphs. We first revisit algorithms IncrFlow and BinaryDC, identifying the repeated solving of large max-flow instances as the primary performance bottleneck. To address this issue, we propose a series of new algorithms. The MeanDC algorithm replaces the binary-search-based pivot selection in BinaryDC with a single arithmetic-mean pivot, retaining the same time complexity while significantly reducing the number of max-flow computations. We then propose the core-prepartition technique and develop CoreDC, which leverages the  $k$ -core hierarchy to pre-split the graph and perform localized divide-and-conquer, greatly shrinking the flow networks in practice. For parallel computation, we design a novel heat-extraction technique and propose the parallel algorithm HeatDC and its improved variant CoreHeatDC, which combine heat-extraction with core-prepartition. Extensive experiments on large real-world networks with up to billions of edges demonstrate the effectiveness of our techniques and the high efficiency and scalability of our algorithms.

## REFERENCES

- [1] Esra Akbas and Peixiang Zhao. 2017. Truss-based Community Search: a Truss-equivalence Based Indexing Approach. In *Proc. VLDB Endow.*, Vol. 10, 1298–1309.
- [2] J. Ignacio Alvarez-Hamelin, Luca Dall'Asta, Alain Barrat, and Alessandro Vespignani. 2005. Large scale networks fingerprinting and visualization using the  $k$ -core decomposition. In *NIPS*. 41–50.
- [3] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. 2001. Thread Scheduling for Multiprogrammed Multiprocessors. *Theory Comput. Syst.* 34, 2 (2001), 115–144. <https://doi.org/10.1007/S00224-001-0004-Z>
- [4] Vladimir Batagelj and Matjaz Zaversnik. 2003. An  $O(m)$  Algorithm for Cores Decomposition of Networks. *CoRR cs.DS/0310049* (2003). <http://arxiv.org/abs/cs/0310049>
- [5] Jon Louis Bentley, Dorothea Haken, and James B. Saxe. 1980. A general method for solving divide-and-conquer recurrences. *SIGACT News* 12, 3 (1980), 36–44.
- [6] Alex Beutel, Wanhong Xu, Venkatesan Guruswami, Christopher Palow, and Christos Faloutsos. 2013. CopyCatch: stopping group attacks by spotting lockstep behavior in social networks. In *WWW*. 119–130.
- [7] Ivona Bezáková. 2000. *Compact representations of graphs and adjacency testing*. Master's thesis. Comenius University.
- [8] Markus Blumenstock. 2016. Fast Algorithms for Pseudoarboricity. In *ALENEX*. 113–126.
- [9] Digvijay Boob, Yu Gao, Richard Peng, Saurabh Sawlani, Charalampos E. Tsourakakis, Di Wang, and Junxing Wang. 2020. Flowless: Extracting Densest Subgraphs Without Flow Computations. In *WWW*. 573–583.
- [10] Glencora Borradaile, Jennifer Iglesias, Theresa Migler, Antonio Ochoa, Gordon Wilfong, and Lisa Zhang. 2017. Egalitarian Graph Orientations. *Journal of Graph Algorithms and Applications* 21, 4 (2017), 687–708.
- [11] Glencora Borradaile, Theresa Migler, and Gordon T. Wilfong. 2019. Density decompositions of networks. *J. Graph Algorithms Appl.* 23, 4 (2019), 625–651. <https://doi.org/10.7155/JGAA.00505>
- [12] T.-H. Hubert Chan and Shinuo Ma. 2025. Density Decomposition in Dual-Modular Optimization: Markets, Fairness, and Contracts. *CoRR abs/2505.19499* (2025). <https://doi.org/10.48550/ARXIV.2505.19499> arXiv:2505.19499
- [13] Lijun Chang and Zhiyi Wang. 2022. A Near-Optimal Approach to Edge Connectivity-Based Hierarchical Graph Decomposition. In *Proc. VLDB Endow.*, Vol. 15, 1146–1158.
- [14] Lijun Chang, Jeffrey Xu Yu, Lu Qin, Xuemin Lin, Chengfei Liu, and Weifa Liang. 2013. Efficiently computing  $k$ -edge connected components via graph decomposition. In *SIGMOD*. 205–216.
- [15] Moses Charikar. 2000. Greedy approximation algorithms for finding dense components in a graph. In *APPROX 2000s*, Vol. 1913. Springer, 84–95.
- [16] Chandra Chekuri, Kent Quanrud, and Manuel R. Torres. 2022. Densest Subgraph: Supermodularity, Iterative Peeling, and Flow. In *SODA*. SIAM, 1531–1555.
- [17] Jie Chen and Yousef Saad. 2012. Dense Subgraph Extraction with Application to Community Detection. *IEEE Trans. Knowl. Data Eng.* 24, 7 (2012), 1216–1230.
- [18] Aleksander Bjørn Grodt Christiansen, Ivor van der Hoog, and Eva Rotenberg. 2025. Local Density and Its Distributed Approximation. In *42nd International Symposium on Theoretical Aspects of Computer Science, STACS 2025, Jena, Germany, March 4-7, 2025 (LIPIcs, Vol. 327)*, Olaf Beyersdorff, Michal Pilipczuk, Elaine Pimentel, and Kim Thang Nguyen (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 25:1–25:20. <https://doi.org/10.4230/LIPICS.STACS.2025.25>
- [19] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2003. Reachability and Distance Queries via 2-Hop Labels. *SIAM J. Comput.* 32, 5 (2003), 1338–1355.
- [20] Jonathan Cohen. 2008. Trusses: Cohesive subgraphs for social network analysis. *National security agency technical report* 16, 3.1 (2008), 1–29.
- [21] Maximilien Danisch, T.-H. Hubert Chan, and Mauro Sozio. 2017. Large Scale Density-Friendly Graph Decomposition via Convex Programming. In *Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3-7, 2017*, Rick Barrett, Rick Cummings, Eugene Agichtein, and Evgeniy Gabrilovich (Eds.). ACM, 233–242. <https://doi.org/10.1145/3038912.3052619>
- [22] Shimon Even and Robert Endre Tarjan. 1975. Network Flow and Testing Graph Connectivity. *SIAM J. Comput.* 4, 4 (1975), 507–518.
- [23] Tommaso Fontana, Sebastiano Vigna, and Stefano Zacchiroli. 2024. WebGraph: The Next Generation (Is in Rust). In *Companion Proceedings of the ACM Web Conference 2024*. Association for Computing Machinery, New York, NY, USA, 686–689.
- [24] Marguerite Frank, Philip Wolfe, et al. 1956. An algorithm for quadratic programming. *Naval research logistics quarterly* 3, 1-2 (1956), 95–110.
- [25] Eugene Fratkín, Brian T. Naughton, Douglas L. Brutlag, and Serafim Batzoglou. 2006. MotifCut: regulatory motifs finding with maximum density subgraphs. In *Proceedings 14th International Conference on Intelligent Systems for Molecular Biology*. 156–157.
- [26] Sen Gao, Rong-Hua Li, Hongchao Qin, Hongzhi Chen, Ye Yuan, and Guoren Wang. 2022. Colorful  $h$ -star Core Decomposition. In *ICDE*. 2588–2601.
- [27] Sen Gao, Hongchao Qin, Rong-Hua Li, and Bingsheng He. 2023. Parallel Colorful  $h$ -star Core Maintenance in Dynamic Graphs. In *Proc. VLDB Endow.*, Vol. 16, 2538–2550.
- [28] Andrew V Goldberg. 1984. *Finding a maximum density subgraph*. Technical Report. University of California Berkeley, Berkeley, CA, USA.
- [29] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying  $k$ -truss community in large and dynamic graphs. In *SIGMOD*. 1311–1322.
- [30] Martin Jaggi. 2013. Revisiting Frank-Wolfe: Projection-Free Sparse Convex Optimization. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013 (JMLR Workshop and Conference Proceedings, Vol. 28)*. JMLR.org, 427–435. <http://proceedings.mlr.press/v28/jaggi13.html>
- [31] Patrick M. Jensen, Niels Jeppesen, Anders B. Dahl, and Vedrana Andersen Dahl. 2023. Review of Serial and Parallel Min-Cut/Max-Flow Algorithms for Computer Vision. *IEEE Trans. Pattern Anal. Mach. Intell.* 45, 2 (2023), 2310–2329. <https://doi.org/10.1109/TPAMI.2022.3170096>
- [32] Ruoming Jin, Yang Xiang, Ning Ruan, and David Fuhr. 2009. 3-HOP: a high-compression indexing scheme for reachability query. In *SIGMOD*. 813–826.
- [33] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew Tomkins. 1999. Trawling the Web for Emerging Cyber-Communities. *Comput. Networks* 31, 11-16 (1999), 1481–1493.
- [34] Jérôme Kunegis. 2013. KONECT: the Koblenz network collection (*WWW '13 Companion*). Association for Computing Machinery, New York, NY, USA, 1343–1350.

- <https://doi.org/10.1145/2487788.2488173>
- [35] Rong-Hua Li, Qiushuo Song, Xiaokui Xiao, Lu Qin, Guoren Wang, Jeffrey Xu Yu, and Rui Mao. 2022. I/O-Efficient Algorithms for Degeneracy Computation on Massive Networks. *IEEE Trans. Knowl. Data Eng.* 34, 7 (2022), 3335–3348.
  - [36] Rong-Hua Li, Qiushuo Song, Xiaokui Xiao, Lu Qin, Guoren Wang, Jeffrey Xu Yu, and Rui Mao. 2022. I/O-Efficient Algorithms for Degeneracy Computation on Massive Networks. *IEEE Trans. Knowl. Data Eng.* 34, 7 (2022), 3335–3348. <https://doi.org/10.1109/TKDE.2020.3021484>
  - [37] Rong-Hua Li, Jeffrey Xu Yu, and Rui Mao. 2014. Efficient Core Maintenance in Large Dynamic Graphs. *IEEE Trans. Knowl. Data Eng.* 26, 10 (2014), 2453–2465.
  - [38] Chenhao Ma, Reynold Cheng, Laks V. S. Lakshmanan, and Xiaolin Han. 2022. Finding Locally Densest Subgraphs: A Convex Programming Approach. *Proc. VLDB Endow.* 15, 11 (2022), 2719–2732.
  - [39] David W. Matula and Leland L. Beck. 1983. Smallest-Last Ordering and clustering and Graph Coloring Algorithms. *J. ACM* 30, 3 (1983), 417–427.
  - [40] Lu Qin, Rong-Hua Li, Lijun Chang, and Chengqi Zhang. 2015. Locally Densest Subgraph Discovery. In *KDD*. 965–974.
  - [41] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. <https://networkrepository.com>
  - [42] Barna Saha, Allison Hoch, Samir Khuller, Louiqa Raschid, and Xiao-Ning Zhang. 2010. Dense Subgraphs with Restrictions and Applications to Gene Annotation Graphs. In *RECOMB (Lecture Notes in Computer Science, Vol. 6044)*, Bonnie Berger (Ed.). Springer, 456–472.
  - [43] Ahmet Erdem Sariyüce, Bugra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V. Çatalyürek. 2013. Streaming Algorithms for k-core Decomposition. In *Proc. VLDB Endow.*, Vol. 6, 433–444.
  - [44] Ahmet Erdem Sariyüce and Ali Pinar. 2016. Fast Hierarchy Construction for Dense Subgraphs. In *Proc. VLDB Endow.*, Vol. 10, 97–108.
  - [45] Ahmet Erdem Sariyüce, C. Seshadhri, and Ali Pinar. 2018. Local Algorithms for Hierarchical Dense Subgraph Discovery. In *Proc. VLDB Endow.*, Vol. 12, 43–56.
  - [46] Ahmet Erdem Sariyüce, C. Seshadhri, Ali Pinar, and Ümit V. Çatalyürek. 2015. Finding the Hierarchy of Dense Subgraphs using Nucleus Decompositions. In *WWW*. 927–937.
  - [47] Alexander Shekhovtsov and Václav Hlaváč. 2013. A Distributed Mincut/Maxflow Algorithm Combining Path Augmentation and Push-Relabel. *Int. J. Comput. Vis.* 104, 3 (2013), 315–342. <https://doi.org/10.1007/S11263-012-0571-2>
  - [48] Jessica Shi, Laxman Dhulipala, and Julian Shun. 2021. Theoretically and Practically Efficient Parallel Nucleus Decomposition. In *Proc. VLDB Endow.*, Vol. 15, 583–596.
  - [49] Petter Strandmark and Fredrik Kahl. 2010. Parallel and distributed graph cuts by dual decomposition. In *The Twenty-Third IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2010, San Francisco, CA, USA, 13-18 June 2010*. IEEE Computer Society, 2085–2092. <https://doi.org/10.1109/CVPR.2010.5539886>
  - [50] Nikolaj Tatti. 2019. Density-Friendly Graph Decomposition. *ACM Trans. Knowl. Discov. Data* 13, 5 (2019), 54:1–54:29.
  - [51] Tran Ba Trung, Lijun Chang, Nguyen Tien Long, Kai Yao, and Huynh Thi Thanh Binh. 2023. Verification-Free Approaches to Efficient Locally Densest Subgraph Discovery. In *ICDE*. 1–13.
  - [52] Jia Wang and James Cheng. 2012. Truss Decomposition in Massive Networks. In *Proc. VLDB Endow.*, Vol. 5, 812–823.
  - [53] Elias Samuel Wirth, Thomas Kerdreux, and Sebastian Pokutta. 2023. Acceleration of Frank-Wolfe Algorithms with Open-Loop Step-Sizes. In *International Conference on Artificial Intelligence and Statistics, 25-27 April 2023, Palau de Congressos, Valencia, Spain (Proceedings of Machine Learning Research, Vol. 206)*, Francisco J. R. Ruiz, Jennifer G. Dy, and Jan-Willem van de Meent (Eds.). PMLR, 77–100. <https://proceedings.mlr.press/v206/wirth23a.html>
  - [54] Quan Xue and T.-H. Hubert Chan. 2025. Faster and Efficient Density Decomposition via Proportional Response with Exponential Momentum. *Proc. ACM Manag. Data* 3, 3 (2025), 161:1–161:26. <https://doi.org/10.1145/3725405>
  - [55] Fan Zhang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2017. When Engagement Meets Similarity: Efficient (k, r)-Core Computation on Social Networks. In *Proc. VLDB Endow.*, Vol. 10, 998–1009.
  - [56] Qi Zhang, Rong-Hua Li, Yalong Zhang, Hongchao Qin, and Guoren Wang. 2025. Density decomposition on large static and dynamic graphs: algorithms and applications. *VLDB J.* 34, 6 (2025), 63. <https://doi.org/10.1007/S00778-025-00942-8>
  - [57] Yalong Zhang, Rong-Hua Li, Longlong Lin, Qi Zhang, Lu Qin, and Guoren Wang. 2025. Integral Densest Subgraph Search on Directed Graphs. *Proc. ACM Manag. Data* 3, 3 (2025), 176:1–176:26. <https://doi.org/10.1145/3725313>
  - [58] Yalong Zhang, Ronghua Li, Qi Zhang, Hongchao Qin, Lu Qin, and Guoren Wang. 2024. Efficient Algorithms for Pseudoarboricity Computation in Large Static and Dynamic Graphs. *Proc. VLDB Endow.* 17, 11 (2024), 2722–2734.
  - [59] Yalong Zhang, Rong-Hua Li, Qi Zhang, Hongchao Qin, Lu Qin, and Guoren Wang. 2025. Density Decomposition of Bipartite Graphs. *Proc. ACM Manag. Data* 3, 1 (2025), 30:1–30:25. <https://doi.org/10.1145/3709680>
  - [60] Yalong Zhang, Ronghua Li, Qi Zhang, Hongchao Qin, and Guoren Wang. 2024. Efficient Algorithms for Density Decomposition on Large Static and Dynamic Graphs. *Proc. VLDB Endow.* 17, 11 (2024), 2933–2945.
  - [61] Feng Zhao and Anthony K. H. Tung. 2012. Large Scale Cohesive Subgraphs Discovery for Social Network Visual Analysis. In *Proc. VLDB Endow.*, Vol. 6, 85–96.

## APPENDIX

### A CONVERGENCE OF HEAT-DIFFUSION

We now proceed to prove Theorem 5. First, in Theorem 8, we show that the heat-diffusion process is equivalent to applying the *Frank-Wolfe algorithm*, a widely used method in convex optimization [24, 30], to optimize the following objective:

$$\begin{aligned} \min_{x, y} \quad & \Phi = \sum_{u \in V} x(u)^2 \\ \text{s.t.} \quad & \forall e = (u, v) \in E: y(e, u) + y(e, v) = 1, \\ & \forall e = (u, v) \in E: 0 \leq y(e, u) \leq 1, 0 \leq y(e, v) \leq 1, \\ & \forall u \in V: x(u) = \sum_{e=(u,v) \in E} y(e, u). \end{aligned}$$

Then, in Theorem 9, we prove that the optimal solution  $x^*$  of  $\Phi$  satisfying  $\lceil x^*(u) \rceil = \bar{r}(u)$  for every vertex  $u$ . Finally, in Theorem 10, we establish a guarantee on the convergence rate.

**THEOREM 8.** *In the heat-diffusion iteration, for any vertex  $u$  let  $x^t(u) = h^t(u)/t$ . Then the iteration of  $x^t$  is equivalent to the iteration of  $x$  produced by optimizing  $\Phi$  using the Frank-Wolfe algorithm.*

**PROOF.** We define  $x^t(u) = h^t(u)/t$  and  $\Delta^t(u) = h^t(u) - h^{t-1}(u)$  for every  $u \in V$ . Then  $x^t(u) = \frac{h^{t-1}(u) + \Delta^t(u)}{t} = \frac{t-1}{t} \cdot \frac{h^{t-1}(u)}{t-1} + \frac{1}{t} \Delta^t(u) = (1 - \lambda^t)x^{t-1}(u) + \lambda^t \Delta^t(u)$ , where  $\lambda^t = 1/t$ . In vector form, this can be written as  $x^t = (1 - \lambda^t)x^{t-1} + \lambda^t \Delta^t$ . Thus, the sequence  $x^0, x^1, \dots, x^T$  can be viewed as an iterative optimization process that, at each step  $t$ , moves from  $x^{t-1}$  towards the direction  $\Delta^t$  with step size  $\lambda^t$ .

We next relate this iteration to optimizing  $\Phi$  using the Frank-Wolfe algorithm. In the optimization program, we treat variables  $y$  as the decision variables, while the vertex variables  $x$  are uniquely determined by  $y$  via the constraints  $x(u) = \sum_{e=(u,v) \in E} y(e, u)$  for all  $u \in V$ . The feasible region  $\mathcal{Y}$  in the  $y$ -space is defined by linear constraints  $y(e, u) + y(e, v) = 1$  and box constraints  $0 \leq y(e, u), y(e, v) \leq 1$ , and is therefore a convex polytope. Because  $x$  is an affine function of  $y$  and  $\Phi$  is a quadratic function of  $x$ , the objective  $\Phi$  is convex over  $\mathcal{Y}$ . We can thus apply the Frank-Wolfe algorithm to minimize  $\Phi$ .

For each edge  $e = (u, v)$  we have  $\frac{\partial \Phi}{\partial y(e, u)} = 2x(u)$  and  $\frac{\partial \Phi}{\partial y(e, v)} = 2x(v)$ . Given a current point  $y \in \mathcal{Y}$  and its induced vector  $x$ , the Frank-Wolfe linear subproblem is to find

$$\arg \min_{y' \in \mathcal{Y}} \sum_{e=(u,v) \in E} (2x(u)y'(e, u) + 2x(v)y'(e, v)).$$

This objective can be minimized edge by edge: for each edge  $(u, v)$ , if  $x(u) < x(v)$  we set  $y'(e, u) = 1, y'(e, v) = 0$ , and otherwise we set  $y'(e, u) = 0, y'(e, v) = 1$ . Therefore, the optimal choice for each edge is to assign the unit to the endpoint with the smaller value of  $x$ . This is exactly the same rule used by the heat-diffusion process, where for each edge we add one unit of heat to the endpoint with smaller  $h^{t-1}$ , and hence to the endpoint with smaller  $x^{t-1} = h^{t-1}/(t-1)$ . Consequently, the vector  $\Delta^t$  coincides with the optimal linear subproblem solution  $y'$ . Since  $\lambda^t = 1/t$ , the update  $x^t = (1 - \lambda^t)x^{t-1} + \lambda^t \Delta^t$  is exactly a Frank-Wolfe update with diminishing step size  $\lambda^t = 1/t$  on optimizing  $\Phi$ .  $\square$

Next, in Theorem 9, we show that the IDN corresponds to the ceiling of the optimal solution  $x^*$  of  $\Phi$ .

**THEOREM 9.** *Given the optimal solution  $x^*$  and  $y^*$  of  $\Phi$ , for  $u \in V$ , we have  $\lceil x^*(u) \rceil = \bar{r}(u)$ .*

PROOF. By optimality, we have: for any edge  $e = (u, v) \in E$ , if  $x^*(u) > x^*(v)$ , then  $y^*(e, u) = 0$  and  $y^*(e, v) = 1$  (Property 1); otherwise, we could decrease  $\Phi$  by increasing  $y^*(e, v)$  and decreasing  $y^*(e, u)$ , contradicting the optimality of  $(x^*, y^*)$ .

Fixing an integer  $k$  and define  $H = \{u \in V \mid x^*(u) > k-1\}$ , next we prove  $H = D_k$ . For two disjoint sets  $S, T \subseteq V$ , let  $E_\times(S, T) = \{(u, v) \in E \mid u \in S, v \in T\}$  and  $Y(S, T) = \sum_{e=(u,v), u \in S, v \in T} y^*(e, u)$ . By Property 1, we obtain: for any edge  $e = (u, v) \in E_\times(H, V \setminus H)$  we have  $y^*(e, u) = 0$  and  $y^*(e, v) = 1$  (Property 2). Consequently, for any  $S \subseteq H$ , we have:  $Y(S, (V \setminus H) \setminus S) = 0$  (Property 3). Moreover, from the relationship of  $x$  and  $y$ , we have: for any  $S \subseteq V$ ,  $|E(S)| + Y(S, V \setminus S) = \sum_{u \in S} x^*(u)$  (Property 4).

For any non-empty  $S \subseteq H$ , we have

$$\begin{aligned} |E(H)| - |E(H \setminus S)| &= |E(S)| + |E_\times(S, H \setminus S)| \\ &\geq |E(S)| + Y(S, H \setminus S) \\ \text{(by Property 3)} \quad &= |E(S)| + Y(S, H \setminus S) + Y(S, (V \setminus H) \setminus S) \\ &= |E(S)| + Y(S, V \setminus S) \\ \text{(by Property 4)} \quad &= \sum_{u \in S} x^*(u) \\ &> (k-1) \cdot |S|. \end{aligned}$$

Thus  $H$  satisfies the internally dense condition in Definition 1. The externally sparse condition for  $H$  can be proved in a symmetric way. Therefore  $H$  satisfies both conditions in Definition 1, and we have  $H = D_k$  and  $\bar{r}(u) = \lceil x^*(u) \rceil$  for all  $u \in V$ .  $\square$

Next, we analyze the convergence rate of the heat-diffusion iteration.

THEOREM 10. Let  $x^*$  be the optimal solution of  $\Phi$ . As  $T \rightarrow \infty$ ,  $h^T/T$  converges to  $x^*$ , satisfying the bound  $\|h^T/T - x^*\|_2 \leq \sqrt{\frac{2(1+\ln T) d_{\max} |E|}{T}}$ , where  $d_{\max} = \max_{u \in V} d(u)$  is the maximum degree.

PROOF. Let  $\Phi^t$  and  $\Phi^*$  denote the objective values of the  $t$ -th iteration and the optimal solution, respectively. By the standard guarantee of the Frank-Wolfe algorithm with step size  $1/t$  [30, 53], we have  $\Phi^T - \Phi^* \leq \frac{(1+\ln T)C}{2T}$ , where  $C$  is the curvature constant.

To bound  $C$ , we view  $\Phi$  as a function of the edge variables  $y$  and write  $\Phi(y) = \sum_{u \in V} \left( \sum_{e \ni u} y(e, u) \right)^2 = \|By\|_2^2$ , where  $B$  is the vertex-edge incidence matrix. Then the gradient is  $\nabla \Phi(y) = 2B^T By$  and the Hessian is  $\nabla^2 \Phi(y) = 2B^T B$ . The Lipschitz constant  $L$  of the gradient is upper bounded by the largest eigenvalue of  $2B^T B$ , which equals the largest eigenvalue of  $2BB^T$ . Since  $BB^T$  is diagonal with diagonal entries equal to vertex degree, its largest eigenvalue is  $2d_{\max}$ , and thus  $L = 2d_{\max}$ . All feasible  $y$  satisfy  $0 \leq y(e, u), y(e, v) \leq 1$ , so the diameter of the feasible domain is  $D = \sqrt{2|E|}$ . By standard curvature bounds [30], we have  $C \leq L \cdot D^2$ , therefore  $C \leq 4d_{\max}|E|$ , and hence  $\Phi^T - \Phi^* \leq \frac{2(1+\ln T)d_{\max}|E|}{T}$ .

Since  $\Phi$  is 2-strongly convex in  $x$ , it follows that  $\|x^T - x^*\|_2^2 \leq \Phi^T - \Phi^* \leq \frac{2(1+\ln T)d_{\max}|E|}{T}$ . By Theorem 8, we have  $x^T = h^T/T$ , and therefore  $\|h^T/T - x^*\|_2 \leq \sqrt{\frac{2(1+\ln T) d_{\max} |E|}{T}}$ .  $\square$