SIGHP: Scalable Information-Guided Hypergraph Partitioner

Anonymous Author(s)
Submission Id: 43

ABSTRACT

Efficient exploration and extraction of complex higher-order relationships within large hypergraphs are critical in the era of massive data proliferation. To tackle the challenges of distributed hypergraph computing, a scalable hypergraph partitioner is essential. Existing research has primarily focused on extending conventional graph partitioners to hypergraph versions but often overlooks the distinctive characteristics, particularly in (hyper)edge size. This work builds upon a novel observation revealing a strong correlation between partitioning metrics and intra-partition hyperedge information influenced by hyperedge size. We extend Shannon's information theory to hypergraphs, providing a novel perspective for hypergraph partitioning. Based on that, we introduce a scalable information-guided hypergraph partitioner, called *SIGHP* [1]. SIGHP stands out as the first partitioner to integrate the strengths of information theory and (hyper)graph theory, showcasing remarkable scalability and quality for hypergraph partitioning. Extensive experiments are done on real hypergraphs and downstream tasks to demonstrate the superiority of SIGHP. The results show that SIGHP not only reduces communication cost and memory overhead by up to 40% and 60%, but also achieves up to 5× enhancement in partitioning efficiency compared to *state-of-the-art* hypergraph partitioners.

KEYWORDS

Hypergraph Partitioning; Scalability; Information Theory; Distributed Processing

1 INTRODUCTION

Graphs serve as a ubiquitous abstraction for modeling relationships (edges) among entities (vertices) in analytical tasks of various domains, including social networks [4], traffic network [7], and bioinformatics [43]. However, graphs are known to have limits in modeling intricate higher-order relationships, as edges only represent pairwise relationships between vertices [23, 26]. To address this limitation, hypergraphs extend edges to hyperedges, which capture relationships involving multiple vertices simultaneously. Hyperedges, formed by connecting arbitrary number of vertices, augment the hypergraph's ability to comprehensively represent internal interactions and dependencies between vertices. This is particularly valuable in applications, such as modeling group interactions in social networks [19, 26, 32, 35], analyzing traffic flow with transportation junctions [38, 41], and depicting gene interaction networks in bioinformatics [11, 37, 39].

Within the massive proliferation of data, dealing with large-scale real-world hypergraphs introduces challenges for single-machine systems [8, 32, 40]. To alleviate the computation overhead and storage expense on individual nodes while effectively utilizing the cluster computational resources, distributed hypergraph processing has become a prominent solution [18, 20]. Partitioning algorithms play a critical role in minimizing inter-partition communication, ensuring a balanced computational workload among partitions, and maximizing the utilization of distributed processing capabilities.

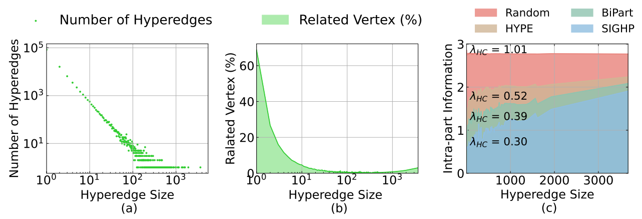


Figure 1: Hyperedge Size Distributions for Number of Hyperedges, Related Vertex Percentage, and Intra-part Information

However, Garey et al. [14] demonstrate the NP-hardness of finding a partitioning scheme that satisfies the aforementioned conditions, while Thang et al. [3] assert the absence of approximation algorithms for this problem. Consequently, the recent trend in hypergraph partitioning is shifting towards utilizing heuristic methods to find a feasible partitioning scheme [1, 10, 29].

Existing research into hypergraph partitioning has primarily focused on adapting conventional graph partitioners to address hypergraph partitioning challenges. For instance, KaHyPar [34], which implements a multi-level method [5, 21], utilizes a multi-level coarsening approach [22] to reduce hypergraph size, and applies a fine-grained partitioning algorithm [12]. HYPE [29], a sequential in-memory algorithm, draws inspiration from NE [42], using a simple neighbor-based heuristic to enhance partitioning efficiency. In scenarios with constrained computational resources, FREIGHT [10] extends the Fennel [36] algorithm from streaming graph partitioning to hypergraphs.

Although there are similarities between graph partitioning and hypergraph partitioning, a crucial distinction lies in the size of (hyper)edges. Graph involves edges connecting only pairs of vertices, whereas hypergraph edges can accommodate an arbitrary number of vertices. Additionally, real-world hypergraphs exhibit significant variations in hyperedge size, typically following a power-law distribution, as illustrated in Figures 1 (a). Furthermore, Figure 1 (b) shows the percentage of vertices covered by hyperedges of different sizes. Numerous small hyperedges collectively connect the vast majority of vertices in the hypergraph, indicating the significant role of small hyperedges in the overall hypergraph structure. This notable difference highlights the importance of considering the impact of hyperedge size on the partitioning quality. Building upon this perspective, we delve into an analysis of partition results obtained from different partitioners, seeking to clarify the impact of hyperedges on the partitioning quality. In Figure 1 (c), a notable observation emerges, highlighting a substantial correlation

[1]Our source code is available online (https://github.com/SIGHyperPartitioner/SIGHP).

between intra-partition hyperedge information[2] and the partitioning quality indicated by λ_{HC} metric (lower is better with detailed explanations provided in Section 2). Higher intra-partition hyperedge information indicates increased uncertainty for event A: two vertices within the same hyperedge belong to the same partition. This in turn raises the probability of spanning across multiple partitions. It shows that the random partitioner, which randomly assigns vertices to partitions, fails to capture the correlation and thus is inferior in terms of partitioning quality. Recent advances, including HYPE and BiPart, improves partitioning quality by unconsciously and unsubstantially addressing the correlation. However, being agnostic to hyperedge information, these methods struggle to achieve consistently high-quality partitioning. Moreover, current research on partitioning lacks the validation of partitioning schemes in real distributed environments. Performance analysis commonly confines itself to comparative evaluations of structural metrics, such as hyperedge cuts, without offering direct evidence to establish the correlation between the metrics and the actual performance, such as communication volume or processing time. Moreover, there is an oversight in exploring the trade-off between partitioning time and partitioning quality.

To tackle these deficiencies, we study the hypergraph partitioning problem from two perspectives. How does the observation of the correlation between intra-partition hyperedge information and quality metrics benefit the hypergraph partitioning? And what significance does hypergraph partitioning carry for distributed tasks?

Drawing inspiration from the strong correlation observed between partition quality and intra-partition hyperedge information in Figure 1(c), we harness the power of information theory to address hypergraph partitioning problems. In particular, we propose a scalable information-guided hypergraph partitioner, called SIGHP. Bridging hyperedge information and partitioning, SIGHP employs a maximal hyperedge sharing strategy, aiming to minimize intra-partition hyperedge information by prioritizing vertices with the highest number of shared hyperedges within each partition. The efficacy of this strategy lies in reinforcing cohesion within partitions, consequently reducing connectivity between partitions and improving partitioning quality.

To address the influence of hyperedge size on partitioning, we introduce *shared hyperedge information* as a heuristic weight in SIGHP, prioritizing connections of small hyperedges within partitions. The strategy's effectiveness stems from two factors. First, generating additional information during the partitioning process helps reduce the remaining information post-partitioning, thereby decreasing intra-partition hyperedge information. Second, prioritizing small hyperedge connections allows us to leverage their significant role in the power-law distribution of hyperedge degrees. Following extensive optimization efforts, SIGHP demonstrates impressive outcomes, showcasing a noteworthy 5× enhancement in partitioning speed, a substantial 60% reduction in memory usage, and a significant 40% decrease in communication overhead metric in specific scenarios. These enhancements are particularly pronounced when SIGHP is applied to large-scale real-world datasets.

[2] Intra-partition hyperedge information is calculated by $-logP(A)$, where $P(A)$ represents the probability of the event A (vertices within the hyperedge belong to the same partition).

Exploring the impact of partitioner on distributed tasks, we conducted extensive experiments within a distributed environment. To ensure fairness and transparency, we establish a distributed runtime environment using an open-source graph processing engine. We accommodated the unique characteristics of distributed task workloads by implementing two representative hypergraph algorithms. One algorithm (HPR) was designed to exhibit high communication load, while the other (HSSSP) was designed to showcase low communication load. These algorithms functions as downstream hypergraph tasks, allowing us to thoroughly examine the effectiveness of partitioners.

The experimental results validate the correlation between partition metrics and actual communication overhead, underscoring the superiority of the SIGHP partitioner in managing large-scale hypergraphs. To justify the worthiness of partitioning time, we introduce the metric of partition profit ratio. This metric evaluates the benefit gained per unit of partitioning time during task processing, providing a more insightful view on the trade-off between partitioning time and quality.

Our contributions can be summarized as follows.

- We are the first to leverage the benefits of information theory to tackle the hypergraph partitioning problem, introducing a fresh perspective to this problem.
- SIGHP presents a proficient, high-quality, and cost-effective solution for large-scale hypergraph partitioning, significantly enhancing the efficiency of distributed task execution.
- We are the first to integrate the validation of partitioning in runtime execution of distributed hypergraph tasks, establishing the correlation between partitioning metrics and actual communication costs. Additionally, we introduce a novel metric that assesses the partitioning profits, offering valuable insights into the trade-off between partitioning time and the benefits gained during task processing.

2 PRELIMINARY

2.1 Concepts of Hypergraphs

Let $H = (V, E)$ be a hypergraph, where $V = \{v_1, v_2, ..., v_n\}$ is a set of vertices and $E = \{e_1, e_2, ..., e_m\}$ is a set of hyperedges. Different from conventional graphs, a hyperedge connects multiple vertices, implying that a hyperedge is essentially a subset of the vertex set V (i.e., $e_i \subseteq V$ and $e_i \in E$). The volume of hypergraph H is denoted as $|H|$, calculated by summing up the sizes of all hyperedges: $|H| = \sum_{e_i \in E} |e_i|$. For a given vertex v_i, $E_r(v_i)$ represents the set of hyperedges relevant to v_i. For a vertex set $V' = \{v'_1, .., v'_i\}$, $E_r(V')$ denotes the union of the sets $E_r(v'_1), ..., E_r(v'_i,)$, expressed as $E_r(V') = E_r(v_1) \cup ... \cup E_r(v_j)$. For a further understand the relevant concept of hypergraph, we provide an example of hypergraph in Appendix A.2

2.2 Problem Statement

The problem of k-way hypergraph partitioning for $H = (V, E)$ refers to finding an allocation function for a partition scheme $P = \{p_1, p_2, ..., p_k\}$, where for any $v_i \in V$, there exists $p_j \in P$ such that $v_i \in p_j$, and satisfying the following constrains: $\cup_{i=1}^{k} p_i = V$ and $p_i \cap p_j = \emptyset$ $(i \neq j)$. Additionally, the partitioning become ϵ-balanced if it adheres to:

$$\max_{p_i \in P}\{|p_i|\} \leq (1+\epsilon)\lceil \frac{|V|}{k} \rceil \tag{1}$$

The partitioning quality metric *hyperedge-cut* (also known as $k-1$ *Metric* [27, 34]) is defined as:

$$HC = -|E| + \sum_{p_i \in P} |E_r(p_i)| \tag{2}$$

We normalize HC to λ_{HC} to regularize the impact of the dataset scale, where $\lambda_{HC} = \frac{HC}{|E|}$. For a better understanding of HC and the specific calculation steps for λ_{HC}, we provide a detailed examples in Appendix A.2

2.3 Hypergraph Information Theory

We hereby study the hypergraph information theory, building based on Shannon's information theory to gain insights into the structural characteristics and relationships within the hypergraph. Shannon's information theory elucidates the relationship between the probability $\mathbb{P}(A)$ of event A and its information content $\mathbb{I}(A)$. Extending the concept to hypergraphs, we represent $\mathbb{P}(v \in e | v' \in e)$ as the conditional probability that vertex v shares hyperedge e with vertex v' when it is known that $v' \in e$. Thereby, we define the Shared Hyperedge Information for single hyperedge e as follow regards:

Definition 1 (Shared Hyperedge Information for Single Hyperedge)**.** Given vertices v, v', and a hyperedge e, the conditional probability of vertex v shares hyperedge e with vertex v' when it is known that $v' \in e$, is denoted as $\mathbb{P}(v \in e | v' \in e)$. The information content generated when this event occurs as $\mathbb{I}(v \in e | v' \in e)$.

$$\mathbb{I}(v \in e | v' \in e) = -\log\ \mathbb{P}(v \in e | v' \in e) \tag{3}$$

Derived from Shannon's information theory, shared hyperedge information inherits three key properties: non-negativity, monotonicity, and additivity (refer to Appendix A.4). We use the notation $v \in \mathcal{E}$ to denote the event where vertex v is covered by multiple hyperedges $\mathcal{E} = \{e'_1, e'_2, \ldots, e'_k\}$. By leveraging additivity of shared hyperedge information, we extend the shared hyperedge for multiple hyperedges as:

Definition 2 (Shared Hyperedge Information for Multiple Hyperedge)**.** Given vertices v, v' and a hyperedges set $\mathcal{E} = \{e'_1, e'_2, ..., e'_k\}$, the shared hyperedge information for the conditional probability $P(v \in \mathcal{E} \big| v' \in \mathcal{E})$ denoted as $\mathbb{I}_M(v \in \mathcal{E} | v' \in \mathcal{E})$.

$$\mathbb{I}_M(v \in \mathcal{E} | v' \in \mathcal{E}) = \sum_{e'_i \in \mathcal{E}} \mathbb{I}(v \in e'_i | v' \in e'_i) \tag{4}$$

Intuitively, the probability of a vertex being covered by a large hyperedge is higher. Based on the hyperedge size, the probability of a vertex belonging to a hyperedge can be defined as $\mathbb{P}(v \in e) = \frac{|e|}{|V|}$. We can derive the conditional probability as:

$$\mathbb{P}(v \in e | v' \in e) = \frac{\mathbb{P}(v \in e) \cdot \mathbb{P}(v' \in e)}{\mathbb{P}(v' \in e)} = \frac{|e|}{|V|}$$
$$\mathbb{P}(v \in \mathcal{E} | v' \in \mathcal{E}) = \frac{\prod_{i=1}^{k} \mathbb{P}(v \in e'_i) \cdot \prod_{i=1}^{k} \mathbb{P}(v' \in e'_i)}{\prod_{i=1}^{k} \mathbb{P}(v' \in e'_i)} = \frac{\prod_{i=1}^{k} |e'_i|}{|V|^{|k|}} \tag{5}$$

After determining these probabilities, for a partitioning task, we can obtain the corresponding shared information as follow regards.

Definition 3 (Shared Hyperedge Information for Partitioning)**.** Given a vertex $v_j \in V$ and a partiton p_i, the corresponding shared hyperedge information is defined as follow regards:

$$\mathbb{I}_M(v_j | E_r(p_i)) = -\sum_{e_k \in E_r(p_i) \cap E_r(v_j)} \log \frac{|e_k|}{|V|} \tag{6}$$

Where $\mathcal{E}(p_i)$ regard as the union of hyperedges for all vertices within the current partition p_i, denoted as $E_r(p_i) = \cup_{v_j \in p_i} E_r(v_j)$.

3 SCALABLE INFORMATION-GUIDED PARTITIONER

Building upon the observation from Figure 1: the significant variation in hyperedge sizes within the hypergraph and the substantial correlation between intra-partition hyperedge information and the partition quality, We investigate a scalable partitioner from dual vantage points which guided by the hypergraph information theory aforementioned which introduced in Section 2.3.

Intra-Partition Hyperedge Information. Building upon the observed strong correlation between intra-partition hyperedge information and partitioning quality, our aim is to minimize intra-partition hyperedge information in the partitioning results. Intra-partition information reflects the cohesion among hyperedges within a partition, where lower intra-partition information indicates a higher presence of vertices form hyperedges within the partition. Therefore, we prioritize the inclusion of vertices within a higher number of shared hyperedges into the current partition to reduce the intra-partition hyperedge information.

Shared Hyperedge Information. Vertices establish connections through shared hyperedges. Smaller hyperedges have a lower probability of sharing, but once sharing occurs, they generate more information. Intuitively, a larger amount of information generated during partitioning leads to less remaining information after partitioning. Therefore, we treat the shared information as a guiding factor for the impact of shared hyperedges size on partitioning. This not only enhances the cohesion of hyperedges within partitions, but also accounts for the impact of hyperedge size distribution, making the partitioner more inclined to prioritize connections related to small hyperedges.

The SIGHP partitioner takes into the two vantage points into the algorithm implementation. Section 3.1 presents a vanilla information-guided partitioning method, followed by Section 3.2, which explores optimizations applied to enhance SIGHP's efficiency. Finally, Section 3.3 conducts a comprehensive analysis of SIGHP's time and space complexity.

3.1 Information-Guided Partitioner

Algorithm 1 outlines the process of the vanilla algorithm. The algorithm initializes the set of candidate vertices V_c and sets the partition capacity to ensure balance (line 1). It then proceeds to generate k partitions sequentially, with each iteration focusing on constructing one partition (line 2). At each iteration, the partition result p_i is set to ϕ (line 3) before processing the candidate vertices (lines 4-7). The main strategy involves selecting vertices with maximum shared hyperedge information (Equation 6) to allocate them to the current partition (line 5).

Algorithm 1: Information-Guided Partitioner (Vanilla)

```
Input: Hypergraph $H(V, E)$, Partition Number $k$
Output: The partitions $\{p_i\}_{1 \le i \le k}$
1 $V_c \leftarrow V$, $Capacity \leftarrow \lceil \frac{|V|}{k} \rceil$
2 for $i \leftarrow 1$ to $k$ do
3     $p_i \leftarrow \phi$;
4     while $|p_i| < Capacity$ do
5         $v_s \leftarrow \arg\max_{v_j \in V_c} I_M(v_j | E_r(p_i))$
6         $p_i \leftarrow p_i \cup \{v_s\}$, $V_c \leftarrow V_c - \{v_s\}$
7 return $\{p_i\}_{1 \le i \le k}$;
```

Figure 2 shows an iterative operation. Initially, the partition contains vertices v_1 and v_2. After computing the $I_M(v_j|\mathcal{E}(p_i))$ for the remaining vertices, v_3, with the maximum shared hyperedge information at the current iteration, is selected to join the partition. Subsequently, the $I_M(v_j|\mathcal{E}(p_i))$ for the remaining vertices is updated.

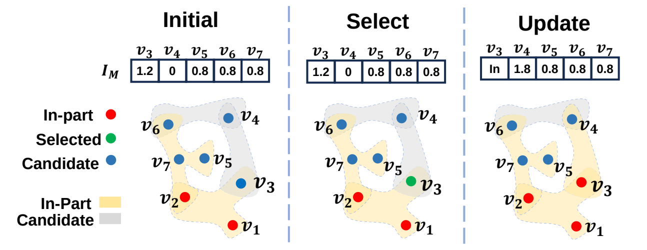


Figure 2: An Example of Iterative Hypergraph Partitioning

Complexity. The time complexity of the vanilla algorithm is $O(|V|^2|E|)$. The main overhead of the algorithm is concentrated in the selection and calculation of nodes with the maximum weight, which needs to be performed $O(|V|)$ times. The time cost per iteration can reach $O(|V||E|)$ in the worst-case scenario.

3.2 Optimizations

In this section, we introduce optimizations aimed at enhancing the efficiency and quality of partitioning. Algorithm 2 outlines the complete workflow of the optimized algorithm, which includes strategies for both improving partitioning efficiency and quality.

3.2.1 *How to improve partitioning efficiency?*

The complexity of the vanilla algorithm primarily arises from the selection of vertices from the candidate set. Thus, our efforts to enhance partitioning efficiency are centered around optimizing the selection and updating processes. In particular, we divide the process into two phase: updating and querying. In the updating phase, we mitigate redundant computations through the use of memorization and incremental update strategies. In the querying phase, we leverage the characteristics of data distribution and employ data structures to expedite the selection process for vertices with the maximum weight.

Updating. During the partitioning process, the shared hyperedge information of candidate vertices undergoes continuous updates. Consequently, we employ memorization and incremental update strategies to minimize redundant computations during the update process. These strategies ensure that updating shared hyperedge information incurs minimal operational costs.

- **Memorization** When a vertex is assigned to the current partition, the expansion of the partition's hyperedge affects the shared hyperedge information weight of neighboring vertices. The power-law distribution of hyperedge size ensures that only a few vertices undergo information updates during the hypergraph contraction process. Moreover, the expected number of updated vertices remains independent of the hypergraph size (refer to proof in Appendix A.5). The memorization strategy thus prevents redundant computations by updating the information weight only when the vertex is involved.
- **Incremental Updates.** The property of information additivity ensures that the change in shared hyperedge information (ΔI_M) is solely associated with the newly added edge e_{add}, regardless of v_j and $E_r(p_i)$.

$$\begin{aligned}\Delta I_M &= I_M(v_j|E_r(p_j) \cup e_{add}) - I_M(v_j|E_r(p_j)) \\ &= -\log \frac{|e_{add}|}{|V|}\end{aligned} \tag{7}$$

Hence, it is sufficient to preprocess ΔI_M, and for all vertices involved in the update, uniform incremental updates can be executed using ΔI_M.

By leveraging the two update strategies, the complexity of the update operation for a single partition can be expressed as $O(|H|)$. However, given that not all hyperedges in a single partition require updates, the actual average complexity is significantly lower than the theoretical upper bound.

Querying. Theorem 2 provides a lower bound for the shared hyperedge information content. Moreover, we proof the sublinear growth of the maximum vertex degree in a power-law distribution (refer to AppendixA.5), thus ensuring that its upper bound remains within a reasonable range. Where c, α are constants and ϕ represents as empty set. The derivation process can be found in the appendix.

$$\begin{aligned}I_M(v_i|E_r(v_j)) &\ge I_M(v_i|\phi) \ge 0 \\ I_M(v_i|E_r(v_j)) &\le c * \sqrt[\alpha+1]{|H|}\end{aligned} \tag{8}$$

These characteristics enhance the efficiency of optimizing a dynamic hash list ($\mathcal{L}$) inspired by pigeonhole problem. The structure of the $\mathcal{L}$, as shown in Figure 3, organizes vertices into buckets according to their information weight. Each bucket employs a hash map as its fundamental data structure to facilitate $O(1)$ queries and modifications. When vertex weights change, the update is executed by relocating the vertex to the bucket corresponding to its new weight. When a vertex weight changes, the update is executed by relocating the vertex to the bucket corresponding to its new weight. Additionally, the bucket automatically expands when its capacity is reached.

To facilitate the swift retrieval of the vertex with the maximum weight, We keep track of the vertices currently presented in which bucket. Additionally, we utilize a pointer P_{max} to identify the bucket containing vertices with the maximum weight ($\mathcal{W}$). The maintenance of P_{max} enables a single query and update to be nearly $O(1)$. For clarity, we offer an example of $\mathcal{L}$ updating in Appendix A.6.

3.2.2 How to improve partitioning quality?

By leveraging the distribution patterns of hyperedge sizes and algorithmic characteristics, we enhance partition quality by sheilding hub hyperedges.

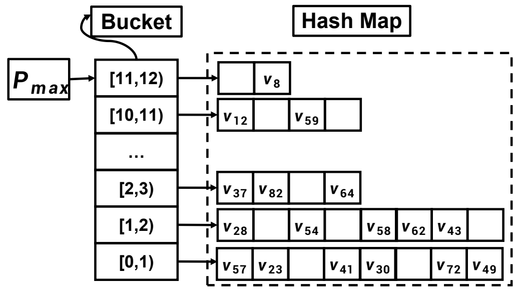

Bucket
Hash Map
P_{max}
[11,12)
[10,11)
...
[2,3)
[1,2)
[0,1)
v_8
v_{12} v_{59}
v_{37} v_{82} v_{64}
v_{28} v_{54} v_{58} v_{62} v_{43}
v_{57} v_{23} v_{41} v_{30} v_{72} v_{49}

Figure 3: Dynamic Hash List Based on Pigeonhole Sorting

This approach not only proves effective in improving partition quality in most scenarios but also leads to a significant reduction in partitioning time.

Hub Hyperedge Shield. A hub hyperedge is defined by its notably larger size compared to others. These hyperedges involving a larger number of nodes are more likely to be added to the current partition, increasing the probability of selecting numerous weakly correlated vertices in subsequent choices. During the initial partitioning phase, the absence of hyperedges with high shared information exacerbates this effect. This not only diminishes partition quality but also leads to typically higher updating costs associated with such hyperedges compared to others. To alleviate the impact of hub hyperedges, we employ the *hub hyperedge shield strategy* to preprocess the hub hyperedges. Taking inspiration from the Pareto principle[9], we define hub hyperedges as the largest several hyperedges $\mathcal{E}_s$, with their cumulative size surpassing a specific percentage γ[3] of entire hypergraph capacity $|H|$, formally:

$$\frac{\sum_{e_i \in \mathcal{E}_s} |e_i|}{|H|} \leq \gamma \tag{9}$$

Algorithm 2: Information-Guided Partitioner (Optimization)

```
   Input: Hypergraph $H(V,E)$, Partition Number $k$
   Output: The partitions $\{p_i\}_{1\le i\le k}$
1  $V_c \leftarrow V, Capacity \leftarrow \lceil \frac{|V|}{k} \rceil$
2  for $i \leftarrow 1$ to $k$ do
3      Initialize weight $\mathcal{W}$ and dynamic hash list $\mathcal{L}$
4      $p_i \leftarrow \phi$
5      while $|p_i| < Capacity$ do
6          $v_s \leftarrow \mathcal{L}.top()$
7          for $e' \in E_r(p_i \cap v_s) - E_r(p_i)$ do
8              if $|e'| > \gamma$ : continue
9              $\delta = -\log \frac{|e'|}{|V|}$
10             for $n_r \in e'$ do
11                 $\mathcal{W}[n_r] \leftarrow \mathcal{W}[n_r] + \delta$
12                 Update $\mathcal{L}$ with new $\mathcal{W}[n_r]$
13         $p_i \leftarrow p_i \cup \{v_s\}$
14 return $\{p_i\}_{1\le i\le k}$
```

3.3 Complexity Analysis

In this section, we study the time and space complexity of the Algorithm 2 and the impact of various optimizations.

[3]In our implementation, we set γ as 0.2, which is discussed in Section 4.5.

Time complexity. For optimized algorithms, the time complexity is composed of two parts: selection and updating. The time complexity of the single selection can be achieved in $O(1)$ using optimized data structures, resulting in the total cost of $O(|V|)$. The updating cost for a single partition p_j can reach $O(|H|)$ in the worst-case scenario, where the partition encompasses all hyperedges. However, in practical execution, especially for power-law hypergraphs, the expected update cost is $O(|E|)$. Thus, the total updating cost can be expressed as $O(k|E|)$ when the hyperedge distribution follows a power-law distribution. In the worst case, it can be represented as $O(k|V||E|)$.

Space complexity. Being an in-memory algorithm, SIGHP utilizes complete graph structural information for partitioning, leading to a fundamental overhead linked to the hypergraph capacity $|H|$, with a worst-case scenario of $O(|E||V|)$. However, as SIGHP relies solely on vertex statistics and connectivity information, the computation process entails minimal additional memory overhead. Therefore, SIGHP tends to be more space-efficient.

4 EXPERIMENTS

4.1 Setup

4.1.1 Datasets. We employed six real-world datasets spanning diverse domains, such as authorship networks (EnWiki), affiliation networks (Orkut), and text networks (Reuter). Table 1 presents the statistics about the hypergraphs. For more detailed information, please refer to the Appendix in Section A.1.

Table 1: Details of Hypergraphs

Datasets	Alias	Category	$\|V\|$	$\|E\|$	$\|H\|$	Size
GitHub [6]	GH	Authorship	56K	120K	440K	4.9M
Reuter [24]	RT	Text	0.8M	0.3M	96M	672M
Trec [31]	TC	Text	1.2M	12M	151M	1.1G
Traker [33]	TK	Hyperlink	12M	27M	140M	1.9G
Orkut [30]	OK	Affiliation	2.7M	8.7M	327M	4.8G
EnWiki [13]	EW	Authorship	8.1M	42.7M	573M	7.6G

4.1.2 Baselines. It is known that hypergraph partitioning is a critical aspect of distributed graph systems, influencing workload distribution, communication efficiency, scalability, and overall performance of downstream distributed tasks. We compare the performance of SIGHP with four state-of-the-art competitors, KaHyPar [34], BiPart [27], HYPE [29], MinMax [1], and Random, as baselines.

4.1.3 Metrics. The comprehensive evaluation of hypergraph partitioning includes both runtime and quality performance.

Runtime Metrics. We monitor and record the overhead of memory/storage consumption, execution time, and networking communication, for the processing of both partitioning and distribute tasks.

Quality Metrics. We adopt the normalized hyperedge-cut metric (λ_{HC}) as the partition quality metric. Furthermore, we assess intra-partition cohesion and inter-partition dissimilarity by utilizing 1-hop neighbor counts and the silhouette coefficient.

Trade-off Metrics. we introduce the partition profit ratio as the metric for evaluating the trade-off between the partition profit and the additional time cost. The metric is computed by dividing the time saved on distributed tasks (compared to random partitioning) by the corresponding partitioning time.

Table 2: Performance Analysis for Running HSSSP, HLP and HPR on a 16-Machines Cluster (k=16), λ_{HC} represent the normalized hyperedge-cut, Communication Recorded in Gigabytes (GB), and Task Runtime Recorded in Seconds (s)

Task	Datasets	MinMax			BiPart			HYPE			SIGHP		
		λ_{HC}	Comm.	Time	λ_{HC}	Comm.	Time	λ_{HC}	Comm.	Time	λ_{HC}	Comm.	Time
HSSSP-1	GH	0.52	0.08	5.8	0.83	0.07	5.6	0.46	0.07	5.9	**0.31**	**0.04**	**5.4**
	RT	7.81	12.5	60.4	2.40	3.7	25	2.39	3.4	22.4	**1.25**	**2.3**	**14.7**
	TC	6.91	8.6	38.4	2.97	4.3	18.8	2.71	3.8	17.6	**1.58**	**3.2**	**13.4**
	TK	1.14	24.1	219.3	0.47	11.8	115.4	0.23	7.2	77.8	**0.13**	**5.7**	**59.9**
	OK	5.64	47.9	192.7	4.04	30.9	126.5	5.59	46.5	165.2	**3.31**	**21.4**	**103.7**
	EW	0.92	67.2	453.3	0.28	25.6	154.7	0.23	23.2	130.1	**0.12**	**19.3**	**90.9**
HSSSP-10	GH	0.52	0.82	59.2	0.83	0.74	57.1	0.46	0.72	58.6	**0.31**	**0.43**	**54.0**
	RT	7.81	127	617	2.40	37.8	253	2.39	33.9	222	**1.25**	**23.7**	**152**
	TC	6.91	92	373	2.97	42	194	2.71	39	181	**1.58**	**33**	**138**
	TK	1.14	237	2215	0.47	121	1193	0.23	73.5	782	**0.13**	**57.2**	**609**
	OK	5.64	483	1899	4.04	314	1302	5.59	455	1662	**3.31**	**209**	**1043**
	EW	0.92	662	4521	0.28	273	1573	0.23	231	1331	**0.12**	**197**	**918**
HLP	GH	0.52	0.10	6.5	0.83	0.09	5.7	0.46	0.09	5.8	**0.31**	**0.06**	**5.3**
	RT	7.81	15.9	96.2	2.40	4.29	32.3	2.39	3.93	37.0	**1.25**	**2.52**	**23.2**
	TC	6.91	10.5	64.9	2.97	4.88	30.9	2.71	4.21	29.1	**1.58**	**3.39**	**19.5**
	TK	1.14	32.1	451	0.47	14.9	219	0.23	8.42	114	**0.13**	**6.20**	**71.9**
	OK	5.64	61.5	306	4.04	41.7	211	5.59	58.7	264	**3.31**	**30.7**	**199**
	EW	0.92	82.9	697	0.28	29.5	199	0.23	26.0	161	**0.12**	**21.3**	**120**
HPR	GH	0.52	0.80	44.5	0.83	0.72	36.5	0.46	0.68	40.4	**0.31**	**0.42**	**35.8**
	RT	7.81	145	1343.3	2.40	29.4	543.7	2.39	25.7	463.3	**1.25**	**11.6**	**283.4**
	TC	6.91	88.2	893.4	2.97	30.0	392.8	2.71	23.3	391.9	**1.58**	**14.9**	**268.6**
	TK	1.14	288	5996.0	0.47	116	2947.2	0.23	53.1	1776.2	**0.13**	**32.2**	**1306.3**
	OK	5.64	499	3756.4	4.04	307	2427.5	5.59	474	3258.4	**3.31**	**182**	**1974.5**
	EW	0.92	677	9881.6	0.28	157	3108.0	0.23	125	2647.9	**0.12**	**75.8**	**1816.2**

4.1.4 Implementation. We utilize PowerGraph [15], a distributed graph processing engine, to examine hypergraph partitioning algorithms' impact on distributed tasks. PowerGraph optimizes processing for natural graphs with power-law distribution, offering both bulk-synchronous Parallel (BSP) and asynchronous models. Compared to alternatives like Pregel [28] and GraphLab [25], PowerGraph enhances scalability for large-scale graph data with specialized methods, while minimizing overhead between cluster nodes compared to Giraph [17] and GraphX [16].

4.1.5 Downstream Distributed Tasks. This study focuses on two key tasks: Hypergraph PageRank (HPR) and Hypergraph Single Source Shortest Path (HSSSP), covering downstream distributed tasks with light and heavy communication loads, respectively.

Hypergraph PageRank (HPR) [2] is an extension of PageRank (PR), originally developed for website ranking. HPR initiates by assigning weights to hyperedges, followed by information aggregation and synchronization. Subsequently, weights are redistributed to vertices, facilitating effective updates and transfers of weights between vertices in the hypergraph. Communication primarily revolves around hyperedge aggregation and synchronization, with active vertices continuously exchanging information until reaching the iteration limit (30 in our experiments) or convergence.

Hypergraph Label Propagation (HLP) is based on the process of label propagation and updating to discover clustering structures in the hypergraph. In the initial stage, vertices are assigned corresponding labels, and in each iteration, vertices propagate their current labels to their neighboring vertices. This process continues iteratively until convergence is achieved. Resonating with the HSSSP algorithm, HLP exhibits a relatively small total communication volume, with a concurrent reduction in active vertices and network communication as iterations advance.

Hypergraph Single Source Shortest Path (HSSSP) is a hypergraph version of the single source shortest path algorithm, calculating the shortest path and distance from a source vertex to others. Similar to SSSP, vertices are considered as neighbors if they share a common (hyper)edge. A source vertex is randomly selected, with its distance set to 0, while others are initialized with infinite distances. Initially, only the source vertex is active, broadcasting the initial distance to its neighbors. As information propagates and distances are updated, both network load and message count increase in the early stages, followed by a subsequent decrease. Since the computation involves finding the shortest paths for individual vertices, while in practice it may be necessary to compute the shortest paths for multiple vertices separately, we have separately conducted HSSSP-1 and HSSSP-10. HSSSP-1 represents the total cost of executing a single HSSSP, while HSSSP-10 represents the total cost of executing 10 independent HSSSPs.

4.1.6 Configuration. All experiments are conducted on Ubuntu 22.04 with 2 x Intel(R) Xeon Platinum 8358 CPUs @ 2.60GHz. The codes for SIGHP and baselines are implemented in C++. We establish a 16-node cluster using Docker to simulate a real-world distributed environment, configuring network bandwidth as 100 Mbps. For baseline algorithms, we utilize their recommended parameter settings. By default, we set the number of threads as 10 for BiPart, ϵ as 0 for load balancing, and γ as 0.2 for SIGHP optimizations. Each reported value is the averages of 5 runs.

4.2 Experiment on Distributed Tasks

4.2.1 ***Task Analysis.*** Table 2 compares our SIGHP against three partitioners: MinMax, BiPart, and HYPE on PowerGraph with a cluster of 16 machines. KaHyPar fails to fulfill the partitioning task for most of datasets within the limited time, hence making it unsuitable for comparison (refer to Section 4.3.1). We measure the hyperedge-cut factor λ_{HC}, communication cost *Comm*. (measured in gigabytes which statistics by distributed processing system), and execution time *Time* (measured in seconds) for running distributed tasks of HPR and HSSSP. SIGHP exhibits a significant advantage across all tested datasets and configurations. Moreover, experimental results indicate that the communication cost and execution time exhibit a strong correlation with the hyperedge-cut factor. While the growth in execution time does not scale proportionally with communication costs.This phenomenon occurs because the overhead of distributed computation consistently occupies a significant portion of time and does not vary with communication volume. We observe inconsistent performance of partitioning algorithms across datasets of different scales. Specifically, despite EW being larger than OK in size, SIGHP and HYPE exhibit superior performance. This suggests that SIGHP has better scalability, leveraging its enhanced ability to explore relationships between hyperedges, thereby effectively improving the performance of computational tasks even as the hypergraph scale increases.

4.2.2 ***Partition Profit.*** We quantify the partitioning profit as the saved execution time of downstream distributed tasks compared to that of the random partitioning. The partition profit ratio is computed by dividing the partitioning partition profit by the corresponding partitioning time (cost). Figure 4 shows the partition profit ratio for processing an HPR task on a 16-node cluster with GH and TC datasets. It shows that SIGHP achieves the highest partition benefit rate among all partitioners. The partition profit ratio approaches almost 40 times, indicating a time savings equivalent to 40 times the partition time in subsequent downstream tasks. In contrast, the profit ratio of KaHyPar is less than 1, suggesting that the overall processing time of KaHyPar is even worse than random partitioning, despite having the least task processing time.

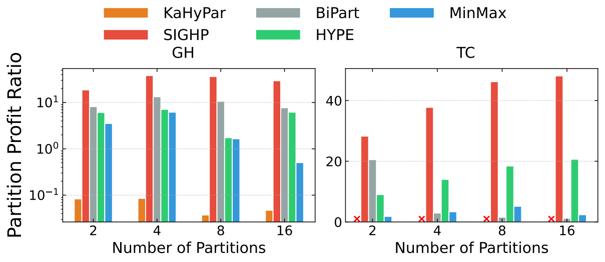
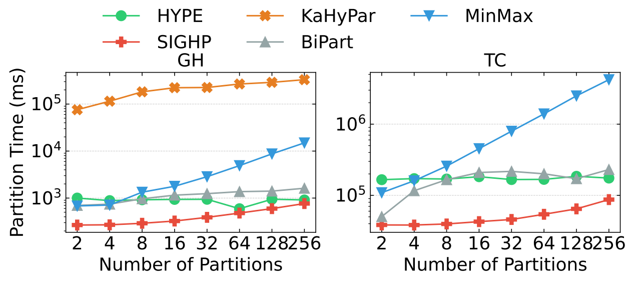
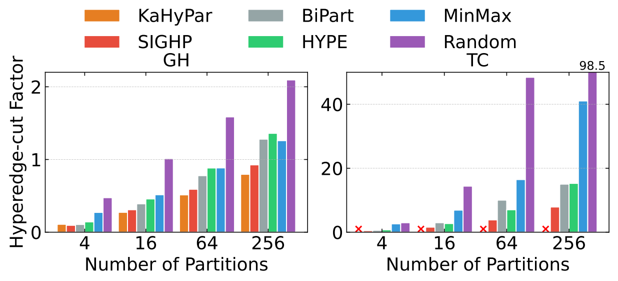


Figure 4: Partition Profit Ratio on GH and TC

In summary, our partitioner, SIGHP, outperforms other existing methods in most scenarios. The superior performance is achieved by not only diminishing communication costs but also reducing task execution time. For example, on the largest dataset TC, SIGHP achieves a 48% reduction in hyperedge-cut factor, a 40% decrease in communication costs, and a 31% reduction in task execution time. The performance improvement in distributed downstream tasks is remarkable. SIGHP exhibits a significant increase in partition profit ratio, exceeding 40 times. This indicates that the performance improvement brought by SIGHP far outweighs the additional overhead.



(a) Partitioning Time



(b) Partitioning Quality

Figure 5: Scalability In Terms Of Efficiency and Quality

4.3 Experiments on Scalability

4.3.1 ***Scalability in Terms of Partition Time & Quality.*** We conduct scalability experiments to show the partitioning performance on both time and quality, by varying the number of partitions from 2 to 256. We compared SIGHP against four other partition algorithms on GH and TC, as shown in Figure 5.

Among all baselines, KaHyPar's partitioning time is two orders of magnitude higher than that of other in-memory algorithms and streaming algorithms, reaching three orders of magnitude in extreme cases. For datasets at the 100MB scale, other algorithms typically complete within a few minutes or even seconds, whereas KaHyPar fails to produce results even within a 24-hour time limit. As the number of partitions increases, most algorithms exhibit an upward trend of partition time (except for HYPE), with noticeable increases in MinMax and BiPart. HYPE, due to its neighbor search strategy, shows partitioning time nearly independent of the number of partitions. Nevertheless, even in such cases, the partitioning time of SIGHP remains significantly lower than that of HYPE.

Moreover, SIGHP also demonstrates superiority in partitioning quality λ_{HC}, specifically the normalized $k-1$ metric, as shown in

Figure 5(b). The result shows that SIGHP outperforms all its competitors except KaHyPar. Despite KaHyPar incurs a time cost that is 1000 times higher, the improvement in quality is not significantly pronounced, compared to SIGHP. Additionally, SIGHP achieves the high-quality partitioning at a lower expense, with a reduction of 40% to 50% in communication cost compared to HYPE and BiPart, all while maintaining shorter partitioning time.

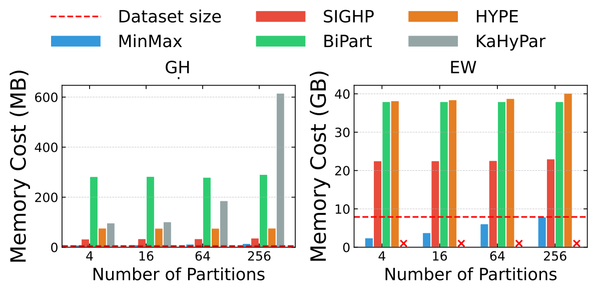


Figure 6: Scalability In Terms Of Memory Consummption

4.3.2 ***Scalability in Terms of Memory Consumption.*** The depicted Figure 6 illustrates the maximum memory consumption of each partitioning algorithm during execution, with the original data size indicated by a red dashed line for comparison. The random partitioning algorithm employs a hash-based approach, leading to almost negligible memory overhead. However, due to its extremely low partitioning quality, further discussion on this algorithm is deemed unnecessary.

Among in-memory algorithms, SIGHP reduces memory overhead by nearly 50% compared to HYPE on EW, and even 60% on GH. It is noteworthy that the memory overhead of in-memory algorithms is almost independent of the number of partitions. While the streaming algorithm MinMax demonstrates an advantage in memory consumption at a low number of partitions, it exhibits a noticeable increase in memory overhead as the number of partitions rises. It is foreseeable that as the number of partitions grows to a certain extent, SIGHP’s memory overhead can be smaller than MinMax.

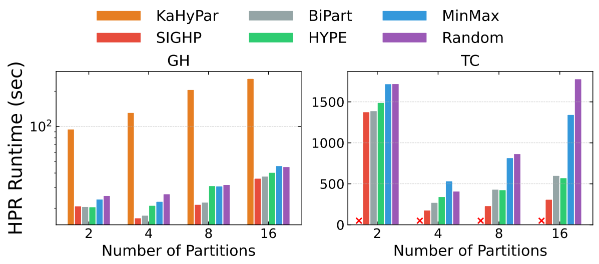


Figure 7: Scalability In Terms Of Total Execution Time

4.3.3 ***Scalability in Terms of Total Execution Time.*** Figure 7 illustrates the impact of graph partitioning on the total task processing time, including both partitioning and task execution time. The results indicate that SIGHP consistently achieves the minimum total processing time in all cases and demonstrates good scalability with respect to the increased number of partitions. Notably, when running on a cluster of 4 machines, SIGHP exhibits a remarkable performance improvement on the test datasets, reducing the total processing time by 75% compared to the case with 2 machines. This observation underscores the effectiveness of scaling out the cluster size in improving the performance of downstream tasks, with SIGHP showcasing good scalability in such scenarios.

4.4 More Analysis on Partitioning Results

In addition to its role in distributed processing, hypergraph partitioning is extensively employed in VLSI design, distributed database optimization, and similarity searching. Consequently, we undertake further analysis on both intra-partition similarity and inter-partition dissimilarity, to get more insights into partitioners.

4.4.1 ***Intra-partition Cohesion.*** Graph cohesion typically refers to the degree of interconnectedness among vertices, whereas each vertex is highly connected to its neighboring vertices. We use 1-hop neighbor count of a partition for representing the density of local graph neighborhood, so as to indicate the graph cohesion. Figure 9 depict the distribution of 1-hop neighbor counts for different partitioning algorithms on GH. The SIGHP algorithm stands out for its notably enhanced 1-hop neighbor relationship compared to other partitioning algorithms, showing an average increase of approximately 30% in neighbor count thus indicating denser interconnectivity among vertices within partitions.

4.4.2 ***Inter-partition Dissimilarity.*** To measure inter-partition dissimilarity, we compute the Jaccard similarity[4] between vertices and all partitions as the feature embedding for each vertex. Subsequently, we utilize t-SNE dimensionality reduction technique for visualization, as shown in the figure 8. Ideally, vertices should exhibit maximum similarity with their own partition and minimum similarity with other partitions. This results in vertices within the same partition being more cohesive and boundaries with vertex from other partitions being clearer. For each partitioner, we also use Silhouette coefficient to quantify the dissimilarity. In Figure 8, SIGHP shows the clearest boundary and the highest Silhouette coefficient, indicating that the highest inter-partition dissimilarity.

4.5 Parameter Analysis

In the folowing discussion, we delve into the the parameter γ which shields the top-k vertices with the maximum size as mentioned in Section 3.2.2. The metric is denoted as $\Delta\lambda_{HC} = \frac{\lambda_{HC}-\lambda'_{HC}}{\lambda_{HC}}$, where λ_{HC} and λ'_{HC} donate the quality metric with and without shielding hub hyperedge, respectively. The acceleration ratio is calculated by dividing the partitioning time before employing hyperedge shielding by the partitioning time after using hyperedge shielding. By varying γ from 0 to 0.3, we draw the following conclusions: as γ increases, partitioning quality initially improves, but beyond a certain threshold, it starts to decline. Also, the partitioning speed radio exhibits nearly linear growth as γ increases. The quality thresholds vary depending on different data scales and graph structures, but they generally indicate significant performance improvement in the range of 0.1 to 0.2. Therefore, setting the parameter γ to 0.2 is a favorable choice for SIGHP in most datasets.

[4]Jaccard similarity between vertex v and partition p calculated by $\frac{|E_r(v)\cap E_r(p)|}{|E_r(v)\cup E_r(p)|}$

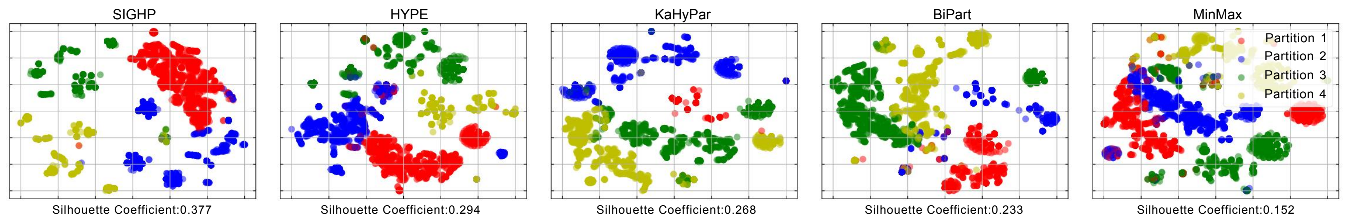


Figure 8: Inter-partition Dissimilarity Analysis about Different Hypergraph Partitioners

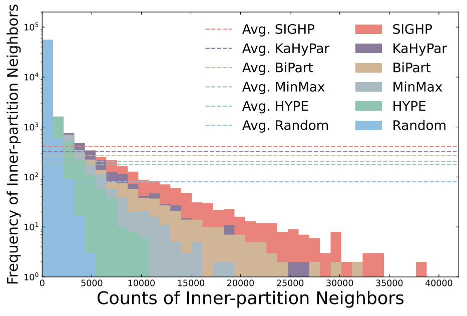


Figure 9: Intra-partition Cohesion Analysis

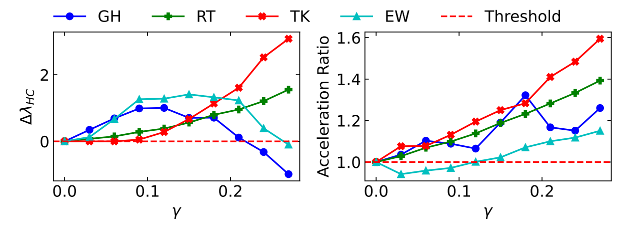


Figure 10: Quality & Acceleration vs. γ

5 CONCLUSION

In the paper, we study the problem of hypergraph partitioning. In particular, we investigate information theory for hypergraph partitioning, by exploring the connection between information metrics and partitioning quality metrics. Building upon the perspective of information theory, alongside the variations in hyperedge size distribution, we introduce a scalable information-guided hypergraph partitioner, called SIGHP. SIGHP offers comparable partitioning efficiency to existing parallel methods, generates high-quality hypergraph partitions, and improves cost-effectiveness with low memory consumption and enhanced partitioning efficiency, making it an excellent solution for managing large-scale hypergraphs.

To underscore the significance of hypergraph partitioning for distributed hypergraph tasks, we conduct extensive experiments to evaluate the performance of different partitioning algorithms in downstream tasks. We propose the metric of partition profit ratio, to balance the cost of partitioning and benefits in distributed task processing. The experiments not only validate the effectiveness of partitioning algorithms but also establish their connection with actual runtime costs. Moreover, structural analysis of partitioning results offers insights for broader applications beyond partitioning.

REFERENCES

[1] Dan Alistarh, Jennifer Iglesias, and Milan Vojnovic. 2015. Streaming Min-max Hypergraph Partitioning. In Advances in Neural Information Processing Systems, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett (Eds.), Vol. 28. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2015/file/83f97f4825290be4cb794ec6a234595f-Paper.pdf

[2] Klessius Berlt, Edleno Silva de Moura, André Carvalho, Marco Cristo, Nivio Ziviani, and Thierson Couto. 2010. Modeling the web as a hypergraph to compute page reputation. Information Systems 35, 5 (2010), 530–543. https://doi.org/10.1016/j.is.2009.02.005

[3] Thang Nguyen Bui and Curt Jones. 1992. Finding good approximate vertex and edge partitions is NP-hard. Inform. Process. Lett. 42, 3 (1992), 153–159. https://doi.org/10.1016/0020-0190(92)90140-Q

[4] William M Campbell, Charlie K Dagli, and Clifford J Weinstein. 2013. Social network analysis with content and graphs. Lincoln Laboratory Journal 20, 1 (2013), 61–81. https://www.researchgate.net/publication/297707303_Social_Network_Analysis_with_Content_and_Graphs

[5] Ümit V Çatalyürek and Cevdet Aykanat. 2011. Patoh (partitioning tool for hypergraphs). In Encyclopedia of parallel computing. Springer, 1479–1487. https://link.springer.com/referencework/10.1007/978-0-387-09766-4

[6] Scott Chacon. 2009. The 2009 GitHub contest. https://github.com/blog/466-the-2009-github-contest

[7] Zhiyong Cui, Kristian Henrickson, Ruimin Ke, and Yinhai Wang. 2019. Traffic graph convolutional recurrent neural network: A deep learning framework for network-scale traffic learning and forecasting. IEEE Transactions on Intelligent Transportation Systems 21, 11 (2019), 4883–4894. https://arxiv.org/abs/1802.07007

[8] Qionghai Dai and Yue Gao. 2023. Large Scale Hypergraph Computation. In Hypergraph Computation. Springer, 145–157. https://link.springer.com/chapter/10.1007/978-981-99-0185-2_8

[9] Rosie Dunford, Quanrong Su, and Ekraj Tamang. 2014. The pareto principle. (2014). https://pearl.plymouth.ac.uk/handle/10026.1/14054

[10] Kamal Eyubov, Marcelo Fonseca Faraj, and Christian Schulz. 2023. FREIGHT: Fast Streaming Hypergraph Partitioning. arXiv (2023). https://arxiv.org/abs/2302.06259

[11] Song Feng, Emily Heath, Brett Jefferson, Cliff Joslyn, Henry Kvinge, Hugh D. Mitchell, Brenda Praggastis, Amie J. Eisfeld, Amy C. Sims, Larissa B. Thackray, Shufang Fan, Kevin B. Walters, Peter J. Halfmann, Danielle Westhoff-Smith, Qing Tan, Vineet D. Menachery, Timothy P. Sheahan, Adam S. Cockrell, Jacob F. Kocher, Kelly G. Stratton, Natalie C. Heller, Lisa M. Bramer, Michael S. Diamond, Ralph S. Baric, Katrina M. Waters, Yoshihiro Kawaoka, Jason E. McDermott, and Emilie Purvine. 2021. Hypergraph models of biological networks to identify genes critical to pathogenic viral response. BMC Bioinformatics 22, 1 (2021), 287. https://doi.org/10.1186/s12859-021-04197-2

[12] C. Fiduccia and R. Mattheyses. 1982. A Linear-Time Heuristic for Improving Network Partitions. In 19th Design Automation Conference. IEEE Computer Society, Los Alamitos, CA, USA, 175,176,177,178,179,180,181. https://doi.org/10.1109/DAC.1982.1585498

[13] Wikimedia Foundation. 2010. Wikimedia downloads. http://dumps.wikimedia.org/

[14] M. R. Garey, D. S. Johnson, and L. Stockmeyer. 1974. Some Simplified NP-Complete Problems. In Proceedings of the Sixth Annual ACM Symposium on Theory of Computing. Association for Computing Machinery, New York, NY, USA, 47–63. https://doi.org/10.1145/800119.803884

[15] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12). USENIX Association, Hollywood, CA, 17–30. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez

[16] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. {GraphX}: Graph processing in a distributed dataflow framework. In 11th USENIX symposium on operating systems design and implementation (OSDI 14). 599–613. https://dl.acm.org/doi/10.5555/2685048.2685096

[17] Minyang Han and Khuzaima Daudjee. 2015. Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. Proceedings of the VLDB Endowment 8, 9 (2015), 950–961. https://dl.acm.org/doi/10.14778/2777598.2777604
[18] Benjamin Heintz, Rankyung Hong, Shivangi Singh, Gaurav Khandelwal, Corey Tesdahl, and Abhishek Chandra. 2019. MESH: A flexible distributed hypergraph processing system. In 2019 IEEE International Conference on Cloud Engineering (IC2E). IEEE, 12–22. https://arxiv.org/pdf/1904.00549.pdf
[19] Shuyi Ji, Yifan Feng, Rongrong Ji, Xibin Zhao, Wanwan Tang, and Yue Gao. 2020. Dual Channel Hypergraph Collaborative Filtering. In Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. Association for Computing Machinery, New York, NY, USA, 2020–2029. https://doi.org/10.1145/3394486.3403253
[20] Wenkai Jiang, Jianzhong Qi, Jeffrey Xu Yu, Jin Huang, and Rui Zhang. 2018. HyperX: A scalable hypergraph framework. IEEE Transactions on Knowledge and Data Engineering 31, 5 (2018), 909–922. https://www.researchgate.net/publication/325918322_HyperX_A_Scalable_Hypergraph_Framework
[21] George Karypis. 1998. hMETIS 1.5: A hypergraph partitioning package. (1998). https://janders.eecg.utoronto.ca/1387/ex2_circuits/manual_hmetis.pdf
[22] George Karypis and Vipin Kumar. 1997. METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices. https://hdl.handle.net/11299/215346
[23] Geon Lee, Jaemin Yoo, and Kijung Shin. 2023. Mining of Real-world Hypergraphs: Patterns, Tools, and Generators. In Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining. Association for Computing Machinery, New York, NY, USA, 5811–5812. https://doi.org/10.1145/3580305.3599567
[24] David D. Lewis, Yiming Yang, Tony G. Rose, and Fan Li. 2004. RCV1: A new benchmark collection for text categorization research. Journal of Machine Learning Research 5 (2004), 361–397. https://www.jmlr.org/papers/volume5/lewis04a/lewis04a.pdf
[25] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. 2014. Graphlab: A new framework for parallel machine learning. arXiv (2014). https://arxiv.org/ftp/arxiv/papers/1408/1408.2041.pdf
[26] Jing Ma, Mengting Wan, Longqi Yang, Jundong Li, Brent Hecht, and Jaime Teevan. 2022. Learning Causal Effects on Hypergraphs. In Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining. Association for Computing Machinery, New York, NY, USA, 1202–1212. https://doi.org/10.1145/3534678.3539299
[27] Sepideh Maleki, Udit Agarwal, Martin Burtscher, and Keshav Pingali. 2021. Bipart: a parallel and deterministic hypergraph partitioner. In Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 161–174. https://dl.acm.org/doi/10.1145/3437801.3441611
[28] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. 135–146. https://dl.acm.org/doi/10.1145/1807167.1807184
[29] Christian Mayer, Ruben Mayer, Sukanya Bhowmik, Lukas EpplQe, and Kurt Rothermel. 2018. HYPE: Massive Hypergraph Partitioning with Neighborhood Expansion. In 2018 IEEE International Conference on Big Data (Big Data). 458–467. https://doi.org/10.1109/BigData.2018.8621968
[30] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2007. Measurement and analysis of online social networks. In Proceedings of the Internet Measurement Conference. https://dl.acm.org/doi/10.1145/1298306.1298311
[31] National Institute of Standards and Technology. 2010. Text REtrieval Conference (TREC) English documents. http://trec.nist.gov/data/docs_eng.html Volume 4 & 5.
[32] Nicolò Ruggeri, Martina Contisciani, Federico Battiston, and Caterina De Bacco. 2023. Community detection in large hypergraphs. Science Advances 9, 28 (July 2023). https://doi.org/10.1126/sciadv.adg9159
[33] Sebastian Schelter and Jérôme Kunegis. 2018. On the ubiquity of web tracking: Insights from a billion-page web crawl. Journal of Web Science 4, 4 (2018), 53–66. http://journals.rudn.ru/web-science/article/view/20296
[34] Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. 2023. High-Quality Hypergraph Partitioning. ACM J. Exp. Algorithmics 27, Article 1.9 (Feb. 2023), 39 pages. https://doi.org/10.1145/3529090
[35] Xiangguo Sun, Hong Cheng, Bo Liu, Jia Li, Hongyang Chen, Guandong Xu, and Hongzhi Yin. 2023. Self-Supervised Hypergraph Representation Learning for Sociological Analysis. IEEE Transactions on Knowledge and Data Engineering 35, 11 (2023), 11860–11871. https://doi.org/10.1109/TKDE.2023.3235312
[36] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. 2014. FENNEL: Streaming Graph Partitioning for Massive Scale Graphs. In Proceedings of the 7th ACM International Conference on Web Search and Data Mining (New York, New York, USA) (WSDM '14). Association for Computing Machinery, New York, NY, USA, 333–342. https://doi.org/10.1145/2556195.2556213
[37] Ramon Viñas, Chaitanya K. Joshi, Dobrik Georgiev, Phillip Lin, Bianca Dumitrascu, Eric R. Gamazon, and Pietro Liò. 2023. Hypergraph factorization for multi-tissue gene expression imputation. Nature Machine Intelligence 5, 7 (2023), 739–753. https://doi.org/10.1038/s42256-023-00684-8
[38] Jingcheng Wang, Yong Zhang, Lixun Wang, Yongli Hu, Xinglin Piao, and Baocai Yin. 2022. Multitask Hypergraph Convolutional Networks: A Heterogeneous Traffic Prediction Framework. IEEE Transactions on Intelligent Transportation Systems 23, 10 (2022), 18557–18567. https://doi.org/10.1109/TITS.2022.3168879
[39] Xuesong Wang, Jian Liu, Yuhu Cheng, Aiping Liu, and Enhong Chen. 2019. Dual Hypergraph Regularized PCA for Biclustering of Tumor Gene Expression Data. IEEE Transactions on Knowledge and Data Engineering 31, 12 (2019), 2292–2303. https://doi.org/10.1109/TKDE.2018.2874881
[40] Yiyang Yang, Sucheng Deng, Juan Lu, Yuhong Li, Zhiguo Gong, Zhifeng Hao, et al. 2021. GraphLSHC: towards large scale spectral hypergraph clustering. Information Sciences 544 (2021), 117–134. https://www.sciencedirect.com/science/article/abs/pii/S0020025520306824
[41] Jaehyuk Yi and Jinkyoo Park. 2020. Hypergraph Convolutional Recurrent Neural Network. In Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. Association for Computing Machinery, New York, NY, USA, 3366–3376. https://doi.org/10.1145/3394486.3403389
[42] Chenzi Zhang, Fan Wei, Qin Liu, Zhihao Gavin Tang, and Zhenguo Li. 2017. Graph Edge Partitioning via Neighborhood Heuristic. In Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. Association for Computing Machinery, New York, NY, USA, 605–614. https://doi.org/10.1145/3097983.3098033
[43] Xiao-Meng Zhang, Li Liang, Lin Liu, and Ming-Jing Tang. 2021. Graph neural networks and their current applications in bioinformatics. Frontiers in genetics 12 (2021), 690049. https://www.frontiersin.org/journals/genetics/articles/10.3389/fgene.2021.690049/full

A APPENDIX

A.1 Dataset Description

The dataset utilized in this study encompasses a variety of real-world data collected from diverse sources. It spans information from multiple domains, including but not limited to text, collaborative networks, the Internet, and citation networks. Such datasets find widespread application in research and analysis within the fields of data mining, machine learning, and artificial intelligence. Below, we provide a detailed introduction to the datasets mentioned in Table 1.

- **GitHub**[6]: The membership network of the software development hosting site GitHub. The network is bipartite and contains users and projects, with links denoting that a user is a member of a project.
- **Reuter**[24]: The bipartite network of story–word inclusions in documents that appeared in Reuters news stories collected in the Reuters Corpus, Volume 1 (RCV1). Left nodes represent stories; right nodes represent words. An edge represents a story–word inclusion.
- **Trec**[31]: The bipartite network of 556,000 text documents from the Text Retrieval Conference's (TREC) Disks 4 and 5, containing 1.1 million words. Each edge represents one document-word inclusion.
- **Traker**[33]: The bipartite network of internet domains and the trackers they contain, such as the Facebook button and Google Analytics. Left nodes are domains and right nodes are trackers, identified also by their domain.
- **Orkut**[30]: The social network of Orkut users and their connections. The network is undirected. The dataset was crawled from the orkut website and may thus be incomplete.

- **EnWiki**[13]: The bipartite edit network of the English Wikipedia. It contains users and pages from the English Wikipedia, connected by edit events. Each edge represents an edit. The dataset includes the timestamp of each edit.

For a better understand the impact of dataset size on partitioning algorithms, we depict the effects of various algorithms as the hypergraph scale grows in Figure 11. We utilize three metrics from left to right in Figure 11: hypergraph size |H|, number of vertices |V|, and number of hyperedges |E|.

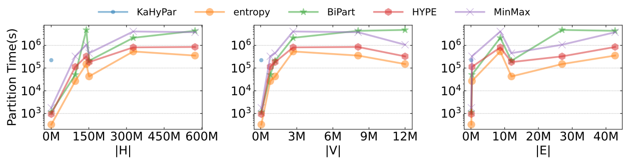


Figure 11: Partition Time with Different Scale Dataset

A.2 Example of Hypergraph

Figure 12 shows three different partition scheme for the same hypergraph with hyperedge set $E = \{e_1, e_2, e_3, e_4\}$, vertices set $V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$ and the $H = \{v_1 \in e_3, v_2 \in e_3, v_2 \in e_4, \ldots\}$. v_2 is covered by hyperedge e_3, e_4, therefore $E_r(v_2) = \{e_3, e_4\}$. For vertices set $V' = \{v_1, v_2, v_3\}$, the $E_r(V') = \{e_2, e_3, e_4\}$.

Hyperedge-Cut is a widely adopted quality metric for hypergraph partitioning. To mitigate numerical discrepancies caused by varying dataset sizes, we propose λ_{HC} for standardizing Hyperedge-Cut. In order to provide readers with a better understanding of the calculation method for Hyperedge-Cut(HC) and tandardizing Hyperedge-Cut(λ_{HC}). We present an example as following:

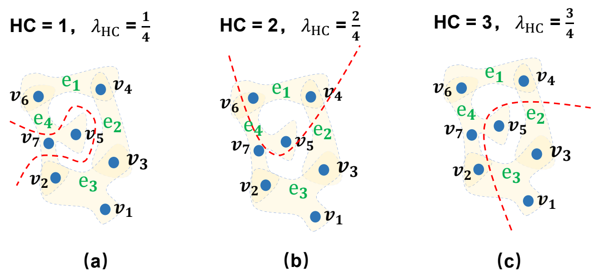

HC = 1, $\lambda_{HC} = \frac{1}{4}$
HC = 2, $\lambda_{HC} = \frac{2}{4}$
HC = 3, $\lambda_{HC} = \frac{3}{4}$
(a)
(b)
(c)

Figure 12: An Example of Hyperedge-Cut and λ_{HC}

In Figure 12, three methods for partitioning the depicted hypergraph are implemented. In Figure 12(a), only e_4 appears in both partitions simultaneously, resulting in HC being 1. Dividing this by the total number of hyperedges yields λ_{HC} as 0.25. Similarly, in Figure 12(b), both e_2 and e_4 appear in both partitions, resulting in HC being 2, corresponding to a λ_{HC} of 0.5. In Figure 12(c), hyperedges e_2, e_3, and e_4 simultaneously appear in both partitions, resulting in HC being 3, with a λ_{HC} value of 0.75.

A.3 Ablation of Efficiency Optimization

In section 3.1 and 3.2 we optimize the efficiency optimization of Algorithm 1 to improve the partition speed and based on these optimization methods we propose Algorithm 2. In this section we will unfold experiments to show the effectiveness of these efficiency optimization, Figure xx represents the difference between the time efficiency of Algorithm 1 and Algorithm 2 on the GitHub graph, it is not difficult to see that Algorithm 2 has got more than the efficiency of Algorithm 1. 2 orders of magnitude improvement over Algorithm 1, which is even more significant for larger graphs with more nodes.

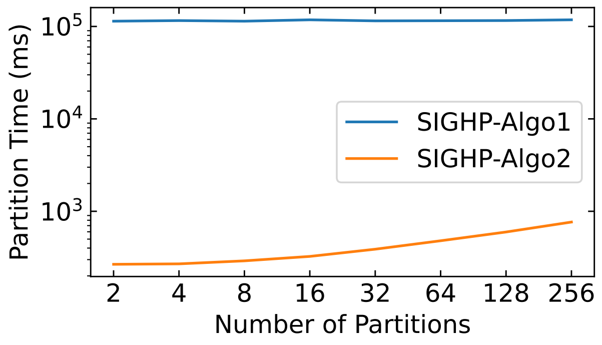


Figure 13: Partition Time of Algorithm 1 and Algorithm 2

A.4 Shared Hyperedge Information Properties

Theorem 1 (The Monotonicity of Shared Hyperedge Information Information). *For given two independent event $A : v, v'$ shared e and $B : v, v'$ shared e' and their probabilities $\mathbb{P}(A), \mathbb{P}(B)$, the shared hyperedge Information $I(A), I(B)$ satisfy:*

$$\begin{aligned} \mathbb{P}(A) < \mathbb{P}(B) &\iff I(A) > I(B) \\ \mathbb{P}(A) = \mathbb{P}(B) &\iff I(A) = I(B) \end{aligned} \tag{10}$$

Proof. Let $f(x) = -\log x$, for any two real numbers a and b where $a < b$, we have:

$$\begin{aligned} f(a) - f(b) &= -\log a + \log b \\ &= \log \frac{b}{a} > \log 1 \end{aligned} \tag{11}$$

Therefore, for any $a < b$, we have $f(a) > f(b)$, and $f(a) = f(b)$ only when $a = b$. The function $f(x)$ is monotonically decreasing. □

Theorem 2 (The Non-negativity of Shared Hyperedge Information). *For arbitrary vertices v, v' and a hyperedge e where v, v' shared e, the shared hyperedge information $I(v \in e | v' \in e)$ satisfies:*

$$I(v \in e | v' \in e) \geq 0 \tag{12}$$

Proof. According Theorem 1, and the domain of probability $x \in [0, 1]$, we have:

$$\begin{aligned} f(x) &\geq f(1) \\ &\geq -\log 1 \end{aligned} \tag{13}$$

Therefore, for any given hypergraph event $v \in e$, the shared hyperedge information is at least non-negative. □

Theorem 3 (The Additivity of Shared Hyperedge Information). *For given two independent belongingness event $A: v, v'$ shared e and $B: v, v'$ shared e'. The shared hyperedge information satisfied:*

$$I(A \cap B) = I(A) + I(B) \tag{14}$$

Proof. For given two independent shared event $A: v, v'$ shared e and $B: v, v'$shared e' and their corresponding probabilities of $[\mathbb{P}(A)$ and $[\mathbb{P}(B)$

$$\begin{aligned} I(A \cap B) &= -\log \mathbb{P}(A \cap B) \\ &= -\log[\mathbb{P}(A) \cdot \mathbb{P}(B)] \\ &= -\log \mathbb{P}(A) - \log \mathbb{P}(B) \\ &= I(A) + I(B) \end{aligned} \tag{15}$$

Therefore, for independent hypergraph event A and B, the additivity of belongingness information is satisfied.

□

A.5 Limited Neighbors

In this section, we prove that in hypergraphs with a power-law distribution, the number of neighbors of a vertex independent with the hypergraph scale $|H|$.

For a given hypergraph $H = (V, E)$, if the hyperedges (or vertices) degree follow a power-law distribution, it can be represented as follows:

$$f(x) = cx^{-\alpha} \tag{16}$$

where $f(x)$ represent the frequency of hyperedge(or vertices) with a degree equal to x, and c denotes a coefficient that needs to be determined. The parameter

$$\alpha \in (1, +\infty)$$

represents the tail exponent of the power-law distribution, determining the rate of decay in the tail of the distribution. In real-world datasets, the typical range for the value of α is within $\alpha \in (1, 3)$. According to the definition of Probability Mass Function, we donate the maximum vertex degree as $|V_{\max}|$, resulting in the following equation:

$$\sum_{x=1}^{|V_{max}|} cx^{-\alpha} = 1 \tag{17}$$

When $V_{\max} \to \infty$, we can approximate the Probability Density Function to a Probability Mass Function as follow:

$$\begin{aligned} \sum_{x=1}^{|V_{max}|} cx^{-\alpha} &= \int_1^{|V_{max}|} cx^{-\alpha} \\ &= -\frac{c}{\alpha+1} \frac{1}{x^{\alpha+1}} \Big|_1^{|V_{max}|} \\ &= 1 \end{aligned} \tag{18}$$

We obtain the coefficient parameter c equal to $\alpha + 1$. Depending on the range of α, the range of c is $(0, 2)$. For modeling the relationship between hypergraph scale H and the vertex size $|V|$, we get the equation based on the definition of hyperedge scale $|H|$:

$$\begin{aligned} |H| &= \int_1^{|V_{max}|} |V| \cdot \frac{\alpha+1}{x^\alpha} \cdot x \\ &= |V| \cdot \frac{\alpha+1}{\alpha} \Big|_{|V_{max}|}^{1} \qquad , when\ |V_{max}| \to \infty \\ &= \frac{\alpha+1}{\alpha} |V| \end{aligned} \tag{19}$$

We argue that the probability of the maximum vertex degree should be at least greater than $\frac{1}{|V|}$. We assume the existence of a threshold t, where $P(|v| > t) < \frac{1}{|V|}$:

$$\begin{aligned} \int_t^\infty \frac{\alpha+1}{x^\alpha} &< \frac{1}{|V|} = \frac{\alpha+1}{\alpha \cdot |H|} \\ \frac{1}{x^{\alpha+1}} \Big|_\infty^t &< \frac{\alpha+1}{\alpha \cdot |H|} \\ t &> \sqrt[\alpha+1]{\frac{\alpha \cdot |H|}{\alpha+1}} \end{aligned} \tag{20}$$

Therefore the maximum possible V_{max} equal to $\sqrt[\alpha+1]{\frac{\alpha \cdot |H|}{\alpha+1}}$. The relationship between hyperedge E_{max} and hypergraph scales $|H|$ is similar. We use α_v and α_e to distinguish the skewness of the vertices degree distribution from the skewness of the hyperedge scale distribution. The expected number of neighbors for vertex v can be expressed as $\mathbb{E}(\mathcal{N}(v)) = \mathbb{E}(E_r(v)) \cdot \mathbb{E}(|e|)$, where $\mathbb{E}(E_r(v))$ represents the expected number of hyperedges associated with vertex v, and $\mathbb{E}(|e|)$ denotes the expected hyperedge scale for each associated hyperedge. The value of $\mathbb{E}(E_r(v))$ can be easily obtained from the above proof as $\frac{\alpha_e+1}{\alpha_e}$. The $\mathbb{E}(\mathcal{N}(v))$ can represent as :

$$\begin{aligned} \mathbb{E}(\mathcal{N}(v)) &= \int_1^{|E_{max}|} |E| \cdot \frac{\alpha_e+1}{x^{\alpha_e}} \cdot \frac{x}{|V|} \\ &= \int_1^{|E_{max}|} \frac{\alpha_e \cdot |H|}{\alpha_e+1} \cdot \frac{\alpha_e+1}{x^{\alpha_e+1}} \cdot \frac{\alpha_v+1}{\alpha_v \cdot |H|} \\ &= \frac{\alpha_e \cdot (\alpha_v+1)}{\alpha_v} \cdot \frac{1}{x^{\alpha_e+1}} \Big|_1^{|E_{max}|} \\ &< \frac{\alpha_e \cdot (\alpha_v+1)}{\alpha_v} \end{aligned} \tag{21}$$

Therefore we proof the limited neighbors for power-law hypergraph.

A.6 Dynamic Hash List Updating

Figure **??** demonstrate a dynamic hash list ($\mathcal{L}$) updating. After updating, v_3 will be assigned to the current partition. In addition to $\mathcal{L}$, here I_M serves as a weight mapping for storing shared hyperedge information, and *Bid* acts as a mapping to store the bucket ID of vertices (e.g., before this update, v_3 was stored in bucket 2).

The update operation can be divided into three steps: 1) Change the status of the selected vertex in *Bid* and I_M. 2) Calculate and update the relevant vertex's I_M. 3) Reallocate the relevant vertices in the $\mathcal{L}$ buckets and update the corresponding *Bid*.

For this example, firstly v_3 change the status as *in* for *Bid* and I_M, and then v_4 update the I_M to 1.8. Lastly remove v_4 form bucket 0 to bucket 3 and update the *Bid* of v_4.