FSM: A Fine-grained Splitting and Merging Framework for Dual-balanced Graph Partition

Chengjun Liu
School of Data Science
Fudan University, China
cjliu23@m.fudan.edu.cn

Zhuo Peng
School of Data Science
Fudan University, China
zpeng21@m.fudan.edu.cn

Weiguo Zheng
School of Data Science
Fudan University, China
zhengweiguo@m.fudan.edu.cn

Lei Zou
Wangxuan Institute of Computer Technology
Peking University, China
zoulei@pku.edu.cn

ABSTRACT

Partitioning a large graph into smaller subgraphs by minimizing the number of cutting vertices and edges, namely cut size or replication factor, plays a crucial role in distributed graph processing tasks. However, many prior works have primarily focused on optimizing the cut size by considering only vertex balance or edge balance, leading to significant workload imbalance and consequently hindering the performance of downstream tasks. Therefore, in this paper, we address the dual-balanced graph partition problem that minimizes the cut size while simultaneously guaranteeing both vertex and edge balance. We propose a lightweight effective two-phase framework, namely fine-grained splitting and merging (FSM), which decomposes the graph into more and smaller partitions and then merges them. FSM offers the flexibility of integrating with various state-of-the-art single-balanced techniques. We develop two efficient algorithms *Fast Merging and Precise Merging* to enable trade-offs between computational efficiency and partitioning quality. Experimental results on large real-world graphs demonstrate that FSM achieves state-of-the-art cut size while maintaining dual balance. The runtime for downstream tasks PageRank, connected component, and diameter estimation, can be reduced by a large proportion, up to 9.43%, 11.35%, and 17.94%, respectively.

PVLDB Reference Format:
Chengjun Liu, Zhuo Peng, Weiguo Zheng, and Lei Zou. FSM: A Fine-grained Splitting and Merging Framework for Dual-balanced Graph Partition. PVLDB, 17(9): 2378 - 2391, 2024.
doi:10.14778/3665844.3665864

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/lcj2021/split-merge-partitioner/.

1 INTRODUCTION

The growing size of graphs presents significant time and memory challenges for algorithms like PageRank [29] and maximal clique enumeration [6, 8]. To address these challenges, graph processing frameworks such as Pregel [25], GraphX [12], and PowerGraph [11] have been developed. These frameworks usually require effective graph partitioning that divides the graph G into a family of groups of vertices and edges. There are two kinds of partitioning paradigms: vertex partitioning and edge partitioning. Vertex partitioning (also known as edge cutting) delivers the vertices of G into pairwise disjoint subsets by cutting the edges. In contrast, edge partitioning (also known as vertex cutting) divides the edges of G into pairwise disjoint subsets by replicating (i.e., cutting) cross-boundary vertices. The number of cutting edges and vertices is called *cut size*.

Table 1: Vertex size imbalance of SOTA partitioners.

Alg. / Graph	hollywood				indochina				arabic			
	$\mathcal{B}_V$	σ_V	$\mathcal{B}_E$	$\mathcal{R}$	$\mathcal{B}_V$	σ_V	$\mathcal{B}_E$	$\mathcal{R}$	$\mathcal{B}_V$	σ_V	$\mathcal{B}_E$	$\mathcal{R}$
NE	1.99	41.23%	1.00	1.53	3.12	73.29%	8.26	1.02	2.31	40.81%	1.00	1.04
HEP-100	1.94	42.61%	1.00	1.55	2.21	36.10%	1.00	1.06	1.90	30.57%	1.00	1.04
METIS	1.77	39.78%	1.03	4.59	2.52	71.93%	1.03	1.09	1.85	44.65%	1.03	1.14

1.1 Motivation

Existing studies [5, 15, 37, 39, 40] have demonstrated that the communication cost is positively correlated with the cut size. Additionally, the deviation between partitioned sets of vertices (known as *vertex balance*) and the deviation between partitioned sets of edges (known as *edge balance*) are essential for achieving workload balance [5]. Thus, lots of efforts have been made to reduce the cut size while ensuring vertex balance [19, 30, 33, 36] or edge balance [26, 31, 38, 39]. However, even the state-of-the-art (SOTA) methods struggle to optimize these three metrics simultaneously. Table 1 presents the performance of three representative partitioners including NE [39], HEP ($\tau = 100$) [26], and METIS [19] over real-world graphs downloaded from WebGraph [2–4], where each graph is partitioned into $p = 32$ parts. Let $\mathcal{B}_V$ (resp. $\mathcal{B}_E$) denote the vertex balance (resp. edge balance), i.e., the fraction of the largest vertex (resp. edge) size over the average vertex (resp. edge) partition size, σ_V denote the coefficient variation of vertex sizes across partitions, and $\mathcal{R}$ denote the replication factor (i.e., the cut size).

As shown in Table 1, we can observe that all three partitioners achieve promising replication factors and edge balances, but (I) exhibit very poor vertex balances, which suggests that they generate

Proceedings of the VLDB Endowment, Vol. 17, No. 9 ISSN 2150-8097.
doi:10.14778/3665844.3665864

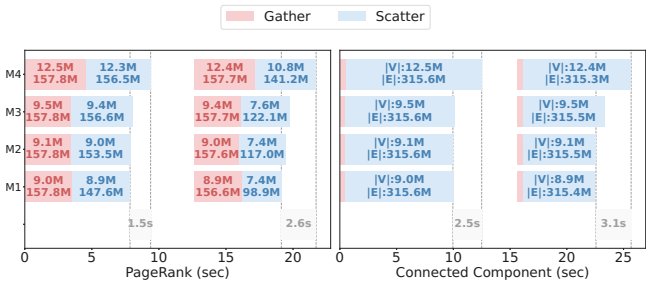


Figure 1: Elapsed time of gather and scatter operations on PowerGraph using NE partitioner.

excessively large partitions; (II) the sizes of these partitions are highly skewed, indicating the existence of partitions with a significantly large number of vertices, as well as partitions with very few vertices. Focusing on only one single balance while minimizing the replication factor can lead to two troublesome problems as follows.
(1) **Terrible Workload Imbalance**. In Bulk Synchronous Parallel (BSP) based systems, a global synchronization checkpoint is set between two iterations. Thus, the machine with the largest workload becomes the performance bottleneck. Figure 1 illustrates the elapsed time of gather and scatter operations on PowerGraph [11] for the top two iterations of the PageRank and Connected Component tasks on four machines (M1, M2, M3, and M4), along with the involved numbers of vertices and edges. While the number of involved edges across the four machines is roughly consistent in the first iteration, there exists a notable discrepancy in elapsed time. Discrepancies in the number of vertices can lead to differences in cache hit rates, as well as disparities in the numbers of involved edges in the subsequent iteration. For example, in the second iteration of PageRank, the discrepancies on the involved edges become significant in the scatter phase, thereby resulting in variations in machine elapsed time.
(2) **Excessive Memory Consumption**. In distributed graph computing frameworks deployed over homogeneous and memory-constrained clusters, each machine needs to maintain necessary information, such as neighbor sets and PageRank values for vertices. The excessively large partitions due to vertex imbalance will significantly increase memory consumption. As for PowerGraph, the partition with the largest number of vertices becomes the memory bottleneck. As depicted in Table 1, for the graph hollywood, the largest partition delivered by METIS has reached 8 times the average size. The machine hosting such a large partition is highly susceptible to encountering memory bottlenecks, which can result in downstream task failures.

Therefore, it is crucial to minimize the replication factor while guaranteeing both vertex and edge balance. However, dual-balanced graph partitioning has not yet received sufficient attention and research despite its importance.

1.2 Challenges and Contributions

Challenge 1: Intractable Complexity. Previous study [39] has established the graph partitioning problem, minimizing the replication factor such that the edge balance is bounded by a constant, which is an NP-hard problem. The inherent complexity coupled with large graphs with billions of vertices and edges already poses significant challenges in terms of time and memory. Dual-balanced graph partitioning problem generalizes the single-balanced graph partition by introducing an additional constraint, making it more difficult to find an optimal solution that satisfies both balances simultaneously. Therefore, the partitioning algorithm itself is expected to be lightweight and efficient. These requirements force us to avoid using overly complex algorithms and redundant data structures.
Challenge 2: Skewed Degrees. Most real-world graphs follow the power-law distribution, revealing a skewed degree distribution that the majority of vertices have low degrees, while only a small subset of vertices are highly connected, known as hub vertices. Hub vertices play a crucial role in influencing the density of partitions, posing a significant challenge in achieving dual balance.

Existing dual-balanced partitioners [1, 24, 40] have made efforts to address these challenges. EBV achieves dual balance by incorporating a scoring function that considers both vertex and edge loads. BPart modifies FENNEL [36] and requires simultaneous optimization of both vertex and edge balance. MDBGP [1] transforms the multi-dimensional balance partitioning problem into a mathematical optimization problem and solves it using gradient descent, but it constrains the number of partitions to powers of two. Beyond that, the optimization process of MDBGP involves multiple rounds of $O(n^2)$ intersection point calculations, which is not feasible for partitioning large graphs. To summarize, these partitioners (I) suffer from poor replication factors, resulting in a substantial communication cost that becomes a bottleneck again, and (II) do not support constraints on vertex-and-edge balance parameters or fail to achieve the desired constraints. There is a notable imbalance in both the vertex and edge dimensions for many graphs.
Contributions. To realize dual-balanced graph partitioning, in this paper, we propose the Fine-grained Splitting and Merging (FSM) framework. The underlying principle behind the FSM framework is to deal with vertex balance and edge balance incrementally, ensuring them one by one. The FSM framework consists of two phases, i.e., fine-grained splitting and subgraph merging.

Specifically, in the fine-grained splitting phase, FSM performs primary exploration of the graph by decomposing it into small-size subgraphs. The aim is to group the highly correlated vertices and edges together, producing a family of fundamental subgraphs. Moreover, in this phase, we can just concentrate on one balance at a time while minimizing replication factors. As a result, the framework becomes highly versatile and powerful, making it possible to integrate with various state-of-the-art single-balance techniques.

In the subgraph merging phase, we assemble the subgraphs produced above into larger partitions. Thus, we can consider the partitioning problem from a broader perspective, enabling us to neutralize the imbalances effectively and enhance the overall performance of the partitioning process. In particular, we formulate a subgraph allocation problem and develop two lightweight and effective merging algorithms, *Fast Merging* and *Precise Merging*.

In summary, our contributions can be summarized as follows:

(1) We formulate the problem of dual-balanced graph partitioning that minimizes the replication factor while both vertex balance and edge balance are guaranteed.

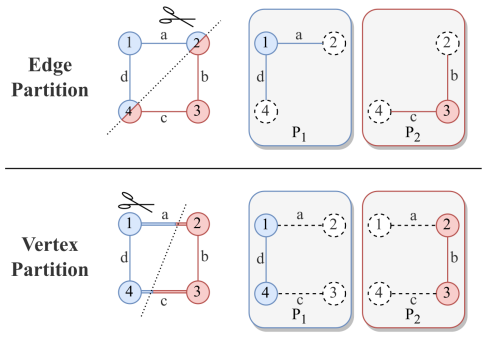


Figure 2: Edge partition and vertex partition.

(2) We develop a lightweight effective two-phase framework, namely fine-grained splitting and merging (FSM), for dual-balanced graph partitioning, that chunks the graph into smaller subgraphs and then merges them to form larger subgraphs.
(3) To enable trade-offs between computational efficiency and partitioning quality, we propose two efficient algorithms *Fast Merging and Precise Merging*. We provide a theoretical optimality analysis of *Fast Merging* and provide the approximation ratio.
(4) We conduct extensive distributed experiments on 11 large graphs against 11 competitors. Compared to the state-of-the-art graph partitioners, FSM exhibits notable efficiency improvements across three distributed graph processing tasks: PageRank, connected component, and approximate diameter.

2 PRELIMINARIES AND OVERVIEW

2.1 Dual-balanced Graph Partitioning

Let $G = (V, E)$ denote an undirected graph, where V and E represent the sets of vertices and edges, respectively. For simplicity, the number of vertices and edges are denoted by n and m, respectively. A p-partitioning divides G into p partitions, where $p \geq 2$ and $p \in \mathbb{N}$.

Definition 1. *(p-Edge Partitioning, also called Vertex Cutting). An edge partitioner divides the edge set E into p pairwise disjoint subsets such that $\bigcup_{i=1}^{p} E_i = E$ and $E_i \bigcap E_j = \emptyset$ for $i \neq j$.*

In edge partitioning, each edge is assigned exactly into one subset. We say a vertex u is a boundary vertex if it is contained in more than one subset, which indicates that u is replicated in these subsets.

Example 1. *As shown at the top of Figure 2, vertices 2 and 4 are boundary vertices. The adjacent edges a and b of vertex 2 are assigned into two different subsets, resulting in a copy of vertex 2.*

Definition 2. *(p-Vertex Partitioning, also called Edge Cutting). A vertex partitioner divides the vertex set V into p pairwise disjoint subsets, i.e. $\bigcup_{i=1}^{p} V_i = V$ and $V_i \bigcap V_j = \emptyset$ for $i \neq j$.*

As shown at the bottom of Figure 2, the vertices {1, 4} and {2, 3} are assigned to two different subsets. Notice that, to guarantee the completeness of the partitions (i.e., without losing any edges), the cutting edges a and c have to be replicated in many applications, leading to vertex cuts as well. Bourse et al. [5] have proved that vertex cuts are smaller than edge cuts on power-law graphs. Therefore, we focus on the widely-used edge partitioning in this paper.

Definition 3. *(Vertex Balance & α-Vertex Balanced Partitioning). The vertex balance of a p-partitioning, denoted by $\mathcal{B}_V$, is defined as $\mathcal{B}_V = \frac{\max_{i=1}^{p} |V_i|}{\sum_{i=1}^{p} |V_i|/p}$. A p-partitioning is α-vertex balanced if $\mathcal{B}_V \leq \alpha$.*

Definition 4. *(Edge Balance & β-Edge Balanced Partitioning). The edge balance of a p-partitioning, denoted by $\mathcal{B}_E$, is defined as $\mathcal{B}_E = \frac{\max_{i=1}^{p} |E_i|}{\sum_{i=1}^{p} |E_i|/p}$. A p-partitioning is β-edge balanced if $\mathcal{B}_E \leq \beta$.*

In the task of edge partitioning, $\mathcal{B}_E$ can be rewritten as $\mathcal{B}_E = \frac{\max_{i=1}^{p} |E_i|}{|E|/p}$ since each edge is just contained in one subset E_i.

Definition 5. *(Replication Factor). The replication factor of a p-partitioning, denoted by $\mathcal{R}$, is defined as $\mathcal{R} = \frac{1}{|V|} \sum_{i=1}^{p} |V_i|$.*

Example 2. *As shown at the top of Figure 2, both partitions contain two edges. Thus, we have the edge balance $\mathcal{B}_E = 1.0$. Vertex 2 and vertex 4 are replicated once each, resulting in a replication factor of $\mathcal{R} = \frac{4+2}{4} = 1.5$.*

Problem Statement. *(Dual-balanced Graph Partitioning). Given a graph G and three parameters p, α, and β, the task of dual-balanced graph partitioning, denoted by MIN-$\mathcal{R}(p, \alpha, \beta)$, is to return a p-edge partitioning of G with the minimum replication factor such that $\mathcal{B}_V \leq \alpha$ and $\mathcal{B}_E \leq \beta$.*

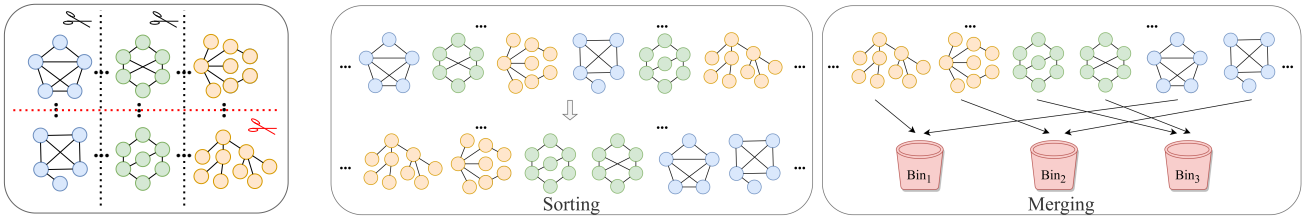
2.2 Overview of the Approach

To reach the objective of minimizing cut size while guaranteeing dual balance, we propose the Fine-grained Splitting and Merging (FSM) framework. FSM adopts an incremental algorithmic approach, by dividing the whole task into two phases as illustrated in Figure 3.

Fine-grained Splitting Phase. In the first *Fine-grained Split* phase, our objective is to obtain more and smaller partitions that have a low cut size and satisfy one single balance constraint at least. Any state-of-the-art edge partitioner or vertex partitioner that achieves a single balance can be used for fine-grained splitting. To maintain allocation information (the subgraph to which a vertex or an edge belongs) and cutting information (the replication status of a vertex or an edge), we develop a succinct data structure called *subgraph information* (denoted as `gInfo`), which can reduce storage costs and accelerate the subsequent merging process.

Subgraph Merging Phase. The core problem addressed in the *Subgraph Merging* phase is to merge the subgraphs obtained in the *Fine-grained Splitting* phase to achieve the final dual-balanced partition. Here, we draw inspiration from the Longest Processing Time (LPT [13]) algorithm. As shown in Figure 3(b), we can abstract the final partitions as bins and shuffle the fine-grained subgraphs. The problem then becomes assigning each subgraph to the appropriate bin. We adopt the *attempt merging* approach for each subgraph to determine its candidate bin. We further design the *attempt merging* process based on efficiency and quality requirements.

Novelty of the approach. The single-balanced algorithms, like NE, HEP, and 2PS [27], only optimize a single dimension, while existing dual-balanced algorithms and systems like BPart [24], MDBGP [1], and EBV [40] demonstrate suboptimal performance in terms of replication factor. FSM addresses these shortcomings by applying an effective two-phase method. The initial phase produces a set of smaller subgraphs with a minimized cut size, optimizing at least one



(a) Fine-grained Splitting Phase

(b) Subgraph Merging Phase

Figure 3: Overview of the FSM framework.

single balance. The second phase merges these subgraphs to enable the dual-balanced guarantee. The two-phase framework is flexible to accommodate powerful splitting or merging strategies. Thus, FSM allows an excellent replication factor while implementing adaptive methods to achieve bounds on two dimensions.

3 FINE-GRAINED SPLITTING

Instead of directly partitioning the graph G into p partitions, we decompose G into more and smaller $p' = k * p$ subgraphs. The intuition is that most existing graph partitioners can achieve good replication factors under a single balance constraint, and we can consider them as small building blocks. These building blocks may have different sizes initially, but the abundance of blocks and their small individual sizes provide us with flexibility and room for optimization in subsequent steps.

3.1 Subgraph Information

To maintain the vertex and edge information of the $k * p$ partitions, we develop a succinct data structure called *subgraph information* (denoted as `gInfo`), which can reduce storage costs and accelerate the subsequent merging process. Specifically, `gInfo` consists of p' dense bitsets (shorted as bitset) named `rep`. When prioritizing edge balance, the `rep` bitset is used to track whether a vertex belongs to a partition. There is also an array named `pvec`, with a length of m for edge partitioners and n for vertex partitioners, which is used to record the initial partition assignment of each vertex or edge.

The `rep` (replication) is typically implemented with a bitset (dense binary string). If the v-th position of `rep[b]` is set to 1, it indicates that vertex v has a replica in P_b. Using a bitset has several advantages. (1) Memory efficiency: Compared to hash-based sets, it can save memory especially when p' is not large (2) Accelerating merging: Using a dense bitset can reduce the constant factor during set merging operations such as bitset OR and bitset popcount (to calculate the number of 1s).

3.2 Prioritizing Edge Balance

To obtain the edge-balance building blocks, we can use SOTA edge partitioners such as NE and HEP. (1) NE [39] is a state-of-the-art edge partitioner known for achieving edge balance and small $\mathcal{R}$, by dividing the partitions into core and boundary vertex sets. It iteratively expands the partitions and prioritizes moving vertices from the boundary set to the core set, particularly those with fewer external connections. Then, it assigns the corresponding edges to the current partition. (2) HEP [26] is a hybrid partitioner that combines NE and HDRF [31]. It partitions high-degree vertices with HDRF, alleviating the skewness of vertices across partitions. This approach also reduces the memory overhead of the adjacency list.

Algorithm 1: Splitting by Edge Partitioner

Input : partition count p, splitting factor k and graph G
Output: subgraph information gInfo (`rep` and `pvec`)

```
1 p' ← p * k
2 pvec ← EdgePart(G, p')
3 foreach e(u, v) ∈ E do
4 |   b ← pvec [e]
5 |   SetBit(rep[b], u, 1)
6 |   SetBit(rep[b], v, 1)
7 return gInfo
```

The detailed algorithm procedure is outlined in Algorithm 1. In lines 3-7, we iterate through each edge $e(u, v)$. We retrieve the subgraph ID b to which e belongs through `pvec`. We set the corresponding bits for vertices u and v in the bitset `rep[b]` to 1 to indicate their presence in g_b.

Example 3. *As illustrated in Figure 4, we begin by obtaining the partition information for each edge through the edge partitioner and storing it in the* `pvec` *array. When processing the edge* $e(6, 7)$*, we first retrieve its assigned subgraph ID, i.e., 2, from* `pvec`*. Then, we set the bits at positions 6 and 7 in the bitset* `rep[2]` *to 1.*

3.3 Overhead Analysis

The time and memory overheads of the *Fine-grained Splitting* phase mainly depend on the partitioner involved. When the splitting factor k is selected, the time and memory overhead of the splitting stage are $T_k(k * p)$ and $M_k(k * p)$, respectively. For each of the $k * p$ partitions, we need to maintain a bitset. When applied to edge partitioners, we track the replication status of vertices, so the length of the bitset is n. The memory overhead of the bitset part is proportional to $O(k * p * n/C)$ and $O(m)$ for `pvec`. C represents the word width which is determined by the underlying hardware and compiler implementation. It should be noted that Algorithm 1 is usually synchronized during partitioning. Thus, the time and memory overheads of these recording steps have already been included in $T_k(k * p)$ and $M_k(k * p)$.

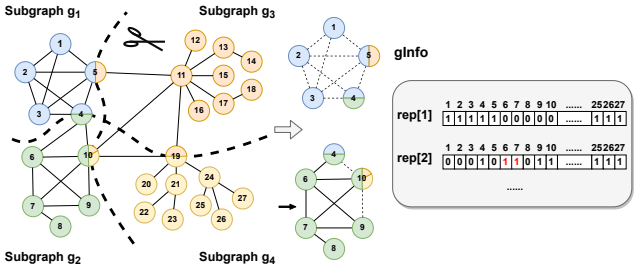


Figure 4: An example of performing splitting.

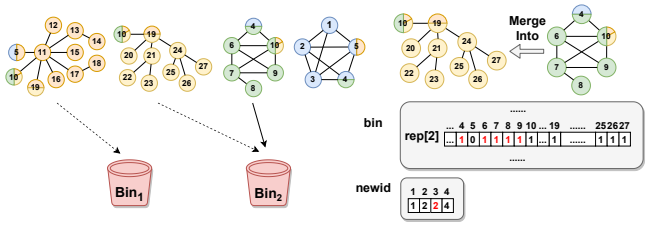


Figure 5: An example of performing merging.

4 SUBGRAPH MERGING

In the *Merging* phase of the FSM framework, the focus shifts from the cutting size and single-balance to the optimization of the other balance metric 'incrementally'. Through the *fine-grained splitting* phase, we obtain $p' = k * p$ small subgraphs (denoted as g). In this phase, the task is to merge these subgraphs into p target partitions $\{P_i\}_{i=1}^{p}$ while achieving a balanced distribution along the other dimension as much as possible. Taking edge partitioners as an example, where the edge balance of the small subgraphs has already been ensured, our goal is to merge exactly k of $\{g_i\}_{i=1}^{p'}$ into one P_j. Thus, we can obtain p partitions with approximately equal numbers of edges, followed by the selection of k subgraphs for each bin.

4.1 Merging Outline

4.1.1 Problem Formulation. We have a set of p' subgraphs $\{g_i\}_{i=1}^{p'}$ along with their information set $\{gInfo_i\}_{i=1}^{p'}$, where each `gInfo` has a bitset `rep`. We want to allocate this subgraph set to p bins to form the final partition $\{P_i\}_{i=1}^{p}$. To maintain the single balance achieved from the *Fine-grained Splitting* phase, the load of each bin is set to k, meaning each bin can accommodate exactly k subgraphs.

Let $\lambda(bin_i)$ denote the popcount of `rep` in the i-th bin, which is resulted from merging k subgraphs. *The objective of subgraph allocation is to distribute $\{g\}$ among the bins in a way that ensures approximately equal popcounts across the bins, while minimizing the maximum popcount among the bins.* Thus, we consider the following minimization problem:

$$\text{minimize } \lambda(bin_M) = \max\{\lambda(bin_1), \lambda(bin_2), ..., \lambda(bin_p)\}.$$

4.1.2 Algorithm Outline.

(1) Subgraph selection. We sort the p' subgraphs in non-ascending order based on their λ. Now subgraph set $\{g\}$ is sorted, so we have $\lambda(g_1) \geq \lambda(g_2) \geq ...\lambda(g_{p'})$. In each iteration, we allocate the first subgraph in the current subgraph set, that is, the subgraph with the largest λ at the current stage (denoted as g_c).

(2) Bin selection. In the second step, we select a candidate bin (bin_{cand}) as the destination for the subgraph g_c.

We maintain a set of currently open bins, called `openbin`, which represents bins that still have available capacity for additional subgraphs. Initially, `openbin` includes all p empty bins and gradually shrinks as subgraphs are assigned to bins during the merge process.

Within `openbin`, we use the *attempt merging* approach to select the candidate bin. This involves iterative merging the pending subgraph with each bin in `openbin` and calculating λ of the resulting bin after the merge. The bin with the smallest λ is chosen as the candidate bin.

(3) Merging. During the merge processing, subgraph g_c is merged into bin_{cand}, leading to a new subgraph $bin_{cand} \cup g_c$. The bin_{cand} is then updated to $bin_{cand} \cup g_c$. We then remove g_c from the subgraph set $\{g\}$ and set the `newid` of g_c to the id of its candidate bin. If bin_{cand} reaches the maximum capacity of k subgraphs, it is closed and removed from `openbin`. We continue this allocation process for each subgraph until all subgraphs are allocated to a bin. Finally, we retrieve the final partitions through `pvec` and `newid`.

Example 4. *As shown in Figure 5, we sort the four subgraphs in non-ascending order based on their λ, resulting in the sorted subgraph set with $\lambda = 11, 10, 6, 5$. We simplify the attempt merging step and directly add λ to determine the size of the new bin. When considering subgraph g_3, its candidate bin is bin_2 because $\lambda(bin_2)$ is currently the smallest. Then we merge bin_2 and g_3 by performing a bitwise OR operation between their corresponding* `rep`. *Finally, we update the entry in* `newid` *for g_3 to indicate its final partition is 2.*

4.2 Merging Algorithm

Based on whether overlap is computed during the bin selection step, we propose two algorithms: Fast Merge and Precise Merge. By offering these two merging options, we provide flexibility to accommodate different requirements and trade-offs between computational efficiency and partitioning quality.

Fast Merge. The Fast Merge algorithm prioritizes speed by simply removing the *attempt merging*, as we assume that there is no overlap in `rep` between the g_c and all the bins. As a result, the *attempt merging* operation, which previously involved bitwise OR operations, can now be simplified to a direct summation of the λs. In this case, we can simplify the bin selection step by selecting the bin with the smallest λ within `openbin`. To further improve speed, we utilize a min-heap to maintain the bin with the smallest λ within `openbin`. The `Pop` operation on the heap (Algorithm 2, lines 4-5) directly produces the candidate bin for merging. If the number of small subgraphs in the newly merged bin does not reach the threshold of k, the bin is placed back into the heap. Otherwise, the merged partition is directly considered as one of the final partitions (Algorithm 2, lines 7-10).

Precise Merge. The *Precise Merge* algorithm is a further refinement of the greedy approach, as it carefully evaluates the overlap when conducting the *attempt merging* step. Specifically, during the allocation of subgraph g_c, it is merged with each bin, and λ of the resulting merged bin is calculated (Algorithm 3, lines 4-6). This

Algorithm 2: Fast Merge

Input : subgraph information gInfo, target partition number p and splitting factor k
Output : partition result

```
1  Initialize min heap Q with p empty nodes
2  Sort(g, rep, non increasing)
3  foreach b ∈ [p'] do
4      candidate ← Q.Pop
5      candidate ← MergeBin(candidate, g_b)
6      newid [b] = candidate.id
7      if candidate.size < k then
8          Q.Push(candidate)
9      else
10         result ← result ⋃ candidate
11 return result
```

Algorithm 3: Precise Merge

Input : subgraph information gInfo, target partition count p and splitting factor k
Output : partition result

```
1  Initialize openbins with p empty bins
2  Sort(g, rep, non increasing)
3  foreach b ∈ [p'] do
4      foreach bin ∈ openbins do
5          newbin ← MergeBin(bin, g_b)
6          Check and update candidate
7      candidate ← MergeBin(candidate, g_b)
8      newid [b] ← candidate.id
9      if candidate.size = k then
10         openbins ← openbins \ candidate
11         result ← result ⋃ candidate
12 return result
```

approach takes into account the potential reduction in λ due to the overlap between g_c and the bins. Once a bin reaches the threshold of k subgraphs, it will no longer be considered in the subsequent merge process and will be included as one of the final partition results (Algorithm 3, lines 9-11).

Overhead Analysis. Both *Fast Merge* and *Precise Merge* require maintaining p bins. Similar to `gInfo`, each bin consists of a bitset ($O(L/C)$). Moreover, we need an array named `newid` of length $k * p$ to record the final partition assignment of each subgraph g. L represents the number of bits in the bitset. If prioritizing edge balance, it is equal to $|V|$, otherwise, it is equal to $|E|$. The bins in *Fast Merge* are maintained using a min heap, and the space complexity is $O(k*p)$. During the entire *Subgraph Merging* process, we first sort the $k * p$ subgraphs, resulting in time complexity of $O(k * p * \log(k * p))$. Then each of the $k * p$ subgraphs needs to merge with its candidate bin once, resulting in a time complexity of $O(k * p * L/C)$ for the entire operation.

Precise Merge performs *attempt merging* on each of the $k * p$ subgraphs. Each *attempt merging* for each subgraph requires a bitset OR operation with all p bins, resulting in a time complexity of $O(p * L/C)$. Hence, the overall time complexity for *Precise Merge* is $O(k * p * \log(k * p) + k * p * (1 + p) * L/C)$. *Fast Merge* efficiently selects candidate bins using a min heap, which can be completed in $O(\log p)$ for each subgraph. Hence, the overall time complexity for *Fast Merge* is $O(k * p * \log(k * p) + k * p * (\log p + L/C))$.

Discussion. As analyzed above, *Fast Merge* runs faster as it has a smaller time complexity. Moreover, it boasts a theoretical guarantee on its approximation rate (see Section 4.3). By utilizing the potential reduction in λ due to the overlap between subgraphs and bins, the merging performance of *Precise Merge* and the downstream tasks based on *Precise Merge* are better than *Fast Merge* in most instances, while with a marginal increase in the merging time. Thus, *Precise Merge* generally emerges as the preferable option, particularly when confronted with intricate bin merging scenarios, especially as the parameter k increases. Nevertheless, we notice that there are cases where *Fast Merge* outperforms *Precise Merge*. The theoretical analysis of *Fast Merge* also provides some hints to *Precise Merge*.

4.3 Theoretical Analysis

We analyze the upper bound of the *Merge* algorithm, that is, the approximation rate compared to the optimal case. To simplify the problem, we do not consider the overlap of bitsets during merging, i.e., we discuss the *Fast Merging* algorithm (which is comparable to *Precise Merging* in terms of merging quality) where we use the λ summation instead of bitset OR. We define $\eta_i = \frac{\lambda(g_i)}{\lambda(g_{p'})}$. Since $\{g\}$ is sorted non-increasingly, we have $\eta = \{\eta_1, \eta_2, ..., \eta_{p'}\}$, $\eta_i \geq 1$ for $\forall i \in [p']$. This optimization objective is also known as the k-partitioning problem (NP-complete when $k = 3$ [10]).

Theorem 1. *When $k = 2$, i.e., when $p' = 2 * p$ subgraphs are merged into p bins, Fast Merge can find the optimal solution.*

Proof. For a more detailed proof, please refer to our technical report. Here, we provide an outline of the proof.

According to the *Fast Merge* algorithm, we place each subgraph g into the current smallest bin. In the case of $k = 2$, the process is simplified to sequentially placing p subgraphs into p bins, followed by placing the remaining p subgraphs in reverse order. Now, consider the scenario where the maximum bin size obtained from the *Fast Merge* algorithm is denoted as ω_0. We will argue by contradiction that there is no other combination that can yield a smaller maximum bin size ω' than ω_0.

Assume that $\omega' = \eta_m + \eta_n$, where m and n are two positive integers representing the indexes of two bins in the new combination. Consider three possible cases for the values of m and n. Case 1:$m, n > p$. Case 2:$m, n \leq p$. Case 3:$m \leq p < n$ and $m + n \neq 2p + 1$. In three cases, we can derive contradictions with the following two conditions. ω' is the maximum bin in the new combination method and ω' is smaller than ω_0. □

Approximation Ratio. Let ω_0 denote the optimal size of the largest bins. The proposed *Fast Merging* algorithm achieves the following approximation ratio:

$$\frac{\omega'}{\omega_0} < 1 + \frac{k-1}{max(p-1+k, p' - (p-1)\eta_1)}.$$

Theorem 2. *Suppose subgraph g_l is the last subgraph placed in the largest bin, with the size of η_l. Let us assume that there are $l (0 \leq l \leq p' - 1)$ subgraphs in total whose vertex sets have sizes greater than or equal to η_l, while the remaining $p' - l$ subgraphs have vertex sets with sizes less than η_l. It holds that $\frac{\omega'}{\omega_0} < (1 + \frac{p-1}{l})$.*

Proof. We define $\varphi_j = \eta_m - \eta_j$, which represents the difference between the total size of the merged subgraph in bin j and the total size of the merged subgraph in the largest bin. We have:

$$\omega_0 = \frac{1}{p}(\sum_{i=1}^{p'} \eta_i + \sum_{j=1}^{p} \varphi_j) \geq \frac{1}{p} \sum_{i=1}^{p'} \eta_i \tag{1}$$

(The equality holds only when $\sum_{j=1}^{p} \varphi_j = 0$, i.e. when the vertex sizes are equal in all bins.) From (1) and $\eta_i \geq 1$, we have:

$$\omega_0 \geq \frac{1}{p} \sum_{i=1}^{p'} \eta_i = \frac{1}{p}(\sum_{\eta_i \geq \eta_l} \eta_i + \sum_{\eta_i < \eta_l} \eta_i) \geq \frac{1}{p}(l * \eta_l + (p' - l)).$$

$$\eta_l \leq \frac{p * \omega - (p' - l)}{l} = \frac{p}{l} * \omega_0 - (\frac{p'}{l} - 1) \tag{2}$$

The equality holds only when there exist l subgraphs with sizes of η_l and $p' - l$ subgraphs with sizes of 1. When allocating g_l to a target bin, there are three cases: (1) No bin has reached k subgraphs and closed. (2) Some bins have reached k subgraphs and closed, but the target bin is still the bin with the smallest total size among all bins. (3) Some bins have reached k subgraphs and closed, but the target bin is not the bin with the smallest total size among all bins.

Now we consider the first two cases because the sizes of the subgraphs to be merged are not significantly different. For $\forall j$, we have $\varphi_j \leq \eta_l$. It holds only when g_l is the last subgraph to be allocated, and before its allocation, the vertex sizes of all subgraphs are evenly distributed, i.e., $l = p' - 1$. Thus, we have:

$$\begin{aligned}\omega' = \frac{1}{p}(\sum_{i=1}^{p'} \eta_i + \sum_{j=1}^{p} \varphi_j) &\leq \frac{1}{p}(\sum_{i=1}^{p'} \eta_i + (p-1)\eta_l + 0) \\ &< (1 + \frac{p-1}{l})\omega_0 - \frac{p-1}{p}(\frac{p'}{l} - 1) \\ &< (1 + \frac{p-1}{l})\omega_0.\end{aligned}$$

This theorem is proved. □

Theorem 3. *For the index l of the subgraph g_l placed last into the largest bin, there is a lower bound: $l \geq p - 1 + k$.*

Proof. Now let us assume that $l < p - 1 + k$. Let bin_M be the largest bin, and let $|bin_M|$ denote the number of subgraphs in bin_M. Since initially all p bins are empty and our greedy strategy prioritizes filling smaller bins first, after distributing the first p subgraphs, each bin contains exactly one subgraph. We have:

$$|bin_M| \leq (l) - p + 1 < (p - 1 + k) - p + 1 < k.$$

At this point, bin_M is already the largest bin. Therefore, the subgraphs with indices $l + 1$ and beyond will be allocated to the remaining bins to balance their sizes with bin_M, and bin_M will not receive any more subgraphs. In other words, the final number of subgraphs in bin_M is less than k, which contradicts the definition of the k-partitioning problem. □

Theorem 4. *For the index l of the subgraph placed last into the largest bin, there is another lower bound: $l \geq p' - (p-1)\eta_1$.*

Table 2: Statistics of Graphs: $\overline{d}$, size, and type denote the average degree, file size, and graph type, respectively.

Name	Graph	$\|V\|$	$\|E\|$	$\overline{d}$	Size	Type
tw	twitter-2010	42 M	1.5 B	70.5	11 G	Soc. Net.
id	indochina-2004	7.4 M	192 M	40.7	1.5 G	Mas. Net.
it	it-2004	41 M	1.1 B	55.0	8.5 G	Mas. Net.
wk	wikipedia	26 M	599 M	46.2	4.5 G	Mas. Net.
uk	uk-2005	39 M	921 M	46.7	6.9 G	Mas. Net.
$u7$	uk-2007-05	105 M	3.7 B	70.7	28 G	Mas. Net.
uu	uk-union	132 M	5.5 B	83.2	41 G	Mas. Net.
sk	sk-2005	51 M	1.9 B	76.2	15 G	Mas. Net.
wb	webbase-2001	116 M	993 M	17.2	7.4 G	Mas. Net.
hw	hollywood-2011	2.0 M	229 M	230.7	1.8 G	Misc. Net.
ar	arabic-2005	23 M	631 M	55.5	4.8 G	Misc. Net.

Proof. Considering the subgraph set sorted in non-increasing order of their sizes, all subgraphs allocated after g_l are used to fill the size gaps between bins. Therefore, we can derive that $\sum_{\eta_i < \eta_l} \eta_i \leq (p-1) * \eta_l$. Since $\sum_{\eta_i < \eta_l} \eta_i \geq p' - l$, we have: $p' - l \leq (p-1)\eta_1$. □

Combining Theorems 2, 3, 4, we conclude that:

$$\frac{\omega'}{\omega_0} < 1 + \frac{p-1}{max(p - 1 + k, p' - (p-1) * \eta_1)}$$

4.4 Adaptive Selection of Splitting Factor

In this section, we discuss how to address the constraints α and β for vertex balance and edge balance, respectively.

Edge balance guarantee. For a specified parameter β, the FSM method initially sets the upper limit of edge capacity for small subgraphs to β times the average number of edges. It then uses a single-balanced partitioner to generate smaller subgraphs.

Vertex balance guarantee. If the vertex balance across the subgraphs generated by the initial fine-grained splitting does not satisfy the demand, it is difficult to achieve the balance effect by using the merge method. FSM achieves the vertex balance by adjusting the splitting factor k. Notice that increasing k improves vertex balance but may also lead to a higher replication factor $\mathcal{R}$. Therefore, for the given parameter α representing vertex balance, we can find the smallest k such that $\mathcal{B}_V \leq \alpha$. This objective can be accomplished through a binary search over the splitting factor. Specifically, if the partitioning result for the current k adheres to the requirement $\mathcal{B}_V \leq \alpha$, the value of k is decreased. Conversely, if the vertex balance requirement is not satisfied, the value of k is increased.

5 EXPERIMENTAL STUDY

5.1 Experimental Setting

Dataset. As shown in Table 2, we use 11 graphs from SNAP [22], WebGraph [2–4], and networkrepository [32] in the experiments.
Methods. We compare the proposed method FSM with 11 partitioners, including streaming, hybrid, and in-memory algorithms. Degree information is taken as input for streaming algorithms.

- DBH [38]: DBH combines random partitioning with degree information by assigning each edge to the hash partition corresponding to the adjacent vertex with the lower degree.
- MDBGP [1]: A method based on gradient descent.

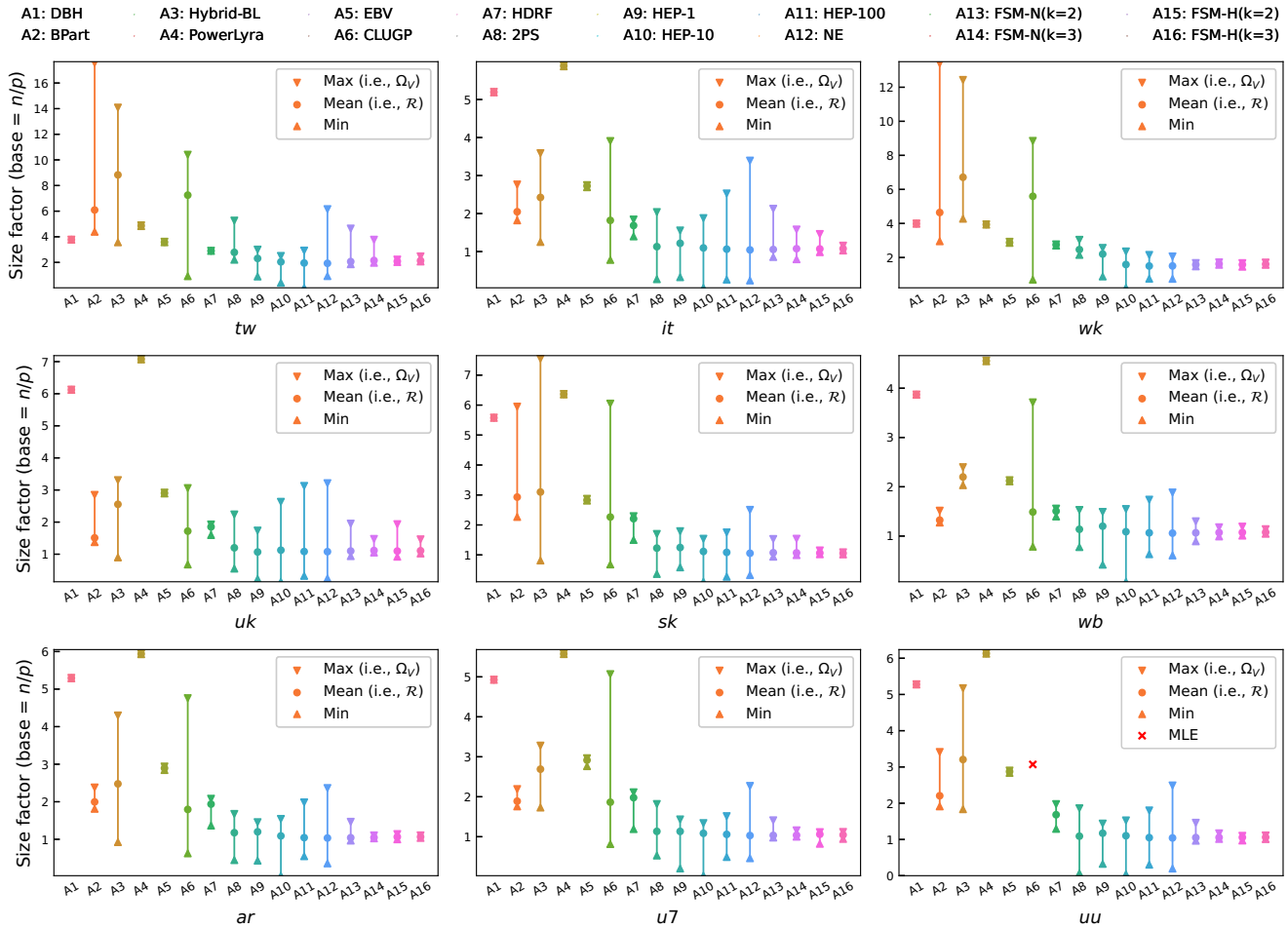


Figure 6: Maximum, mean, and minimum vertex size by 11 partitioners with $p = 32$.

- BPart [24]: BPart relaxes the vertex balance constraint of FENNEL and then optimizes the dual balance based on this relaxation.
- PowerLyra [7]: PowerLyra proposes a Hybrid algorithm that combines random hashing to allocate edges of low-degree and high-degree vertices using different strategies.
- Hybrid-BL (TopoX) [23]: On the basis of Hybrid algorithm, Topox incorporates fusion for low-degree vertices and fission for high-degree vertices, introducing the Hybrid-BL algorithm.
- HDRF [31]: HDRF allocates each edge based on a scoring function that takes into account both vertex degrees and the distribution of vertices and edges in partitions.
- EBV [40]: EBV also utilizes a similar scoring function to allocate edges, but it focuses more on dual balance.
- CLUGP [21]: CLUGP is pipelined into three steps: streaming clustering, cluster partitioning, and partition transformation.
- HEP-1, HEP-10, HEP-100 [26]: HEP partitions high-degree edges by using HDRF and partitions low-degree edges by using NE.
- 2PS [27]: 2PS algorithm consists of two stages: streaming clustering stage and re-streaming stage. 2PS-HDRF is selected as the competitor as it exhibits better partitioning performance.
- NE [39]: NE prioritizes expanding vertices with fewer external connections to generate partitions incrementally.
- FSM-NE: FSM framework utilizes NE as the *Fine-grained Split* partitioner and precise merge, shorted as FSM-N.
- FSM-HEP: FSM framework utilizes HEP-100 as the *Fine-grained Split* partitioner and precise merge, shorted as FSM-H.

Evaluation Metric. We evaluate the performance of the methods through three metrics: the replication factor $\mathcal{R}$, vertex and edge size coefficient variation σ_V and σ_E, and max vertex size factor Ω_V ($\Omega_V = \frac{\max_{i=1}^{p} |V_i|}{n/p}$). The vertex balance $\mathcal{B}_V$ is not reported as it can be represented as $\Omega_V/\mathcal{R}$. We also report the elapsed time over downstream tasks: PageRank [29], connected component computation, and approximate diameter [18].

Reproducibility: We employ the implementation offered by [1, 7, 19, 21, 26, 27, 39] and re-implement EBV, FENNEL using C++. We develop standalone versions of BPart and TopoX using C++.

We set up a distributed cluster with 8 machines, each having Intel(R) Xeon(R) W-2135 CPU @ 3.70GHz and 64 GB of RAM. They are set up with PowerGraph, running at full thread capacity, and connected via Gigabit Ethernet.

Table 3: Time (second) and memory (GB) overhead over graphs uu, u7, and sk with $p = 32$.

		DBH	MDBGP	BPart	Hybrid-BL	PowerLyra	EBV	CLUGP	HDRF	2PS	HEP-1	HEP-10	HEP-100	NE	FSM-H(k=2)	FSM-N(k=2)
uu	Time	395.447	TLE	1669.37	1144.15	470.985	2758.66	MLE	1021.64	1856.99	684.97	365.45	**325.41**	7777.95	644.09	10429.5
	Memory	**0.99**		44.84	107.32	**0.99**	41.78		**0.99**	10.3	17.06	27.56	39.52	98.79	53.22	110.84
u7	Time	513.042	TLE	964.16	647.43	403.585	1746.23	5277.93	748.69	1284.61	420.09	**238.64**	247.48	3629.46	463.33	5971.39
	Memory	**0.79**		30.93	72.29	**0.79**	28.48	31.50	**0.79**	8.27	12.41	20.01	26.80	67.91	36.11	76.32
sk	Time	172.645	TLE	484.36	307.08	227.753	801.82	2438.81	407.79	698.26	190.98	**72.21**	222.75	1869.25	203.66	1965.72
	Memory	**0.38**		15.94	42.85	**0.38**	14.76	16.42	**0.38**	4.07	6.46	11.40	12.91	35.06	17.49	39.37

Table 4: Average σ_V and σ_E of partitioned graphs achieved by different methods.

	DBH	MDBGP	BPart	Hybrid-BL	PowerLyra	EBV	CLUGP	HDRF	2PS	NE	HEP-1	HEP-10	HEP-100	FSM-N(k=2)	FSM-N(k=3)	FSM-H(k=2)	FSM-H(k=2)
σ_V	0.0%	-	20.01%	22.66%	0.1%	0.5%	32.02%	6.7%	29.08%	43.95%	19.88%	27.9%	33.31%	14.24%	6.95%	6.52%	3.67%
σ_E	0.0%		62.04%	32.29%	0.3%	4.46%	21.25%	0.0%	4.94%	12.43%	0.0%	0.0%	0.0%	0.08%	0.15%	0.0%	0.0%

5.2 Overall Partitioning Performance

Figure 6 illustrates the size factor of each partition that is generated by 11 partitioners in detail, where the *size factor* of a partition is defined as the ratio of its vertex size to the average vertex size. Due to space limits, we exclude the results on *id* and *hw*. We evaluate the performance of these partitioners based on the following three metrics. The number of partitions p is set to 32 by default.

5.2.1 Max Vertex Size Factor. The max vertex size factor, i.e., Ω_V, can directly reflect the ***maximum memory overhead and computation workload*** incurred by all machines in the cluster when performing distributed tasks. A good edge partitioner should minimize Ω_V, to ensure that large-scale distributed tasks can be executed smoothly. The large $\mathcal{R}$ of streaming partitioners leads to their Ω_V being inevitably large, as $\Omega_V \geq \mathcal{R}$.

As shown in Figure 6, NE, HEP, and 2PS perform similarly, with a low $\mathcal{R}$ but a high Ω_V. They perform even worse than HDRF on graphs such as *it*. Streaming partitioners such as DBH and EBV are limited to high $\mathcal{R}$, and their Ω_V are also high. We also find that HDRF performs better than other streaming partitioners. Hybrid-BL tends to outperform PowerLyra on most graphs except graphs like *wk* and *tw*. FSMs can achieve the lowest Ω_V among the 11 graphs, especially for FSM-HEP. FSM-HEPs outperform FSM-NE in most cases because HEP outperforms NE in terms of Ω_V in the Fine-grained Split phase initially.

5.2.2 Replication Factor. The *replication factor*, i.e., $\mathcal{R}$, is positively correlated with the ***communication volume*** between clusters. We can observe that NE has the lowest $\mathcal{R}$ on these graphs. The $\mathcal{R}$ of HEP depends on the value of τ. The larger the τ values, the closer HEP is to NE, resulting in a smaller $\mathcal{R}$. Conversely, if the τ is smaller, HEP becomes closer to HDRF, resulting in a larger $\mathcal{R}$. Among re-streaming partitioners, 2PS produces low $\mathcal{R}$, while CLUGP suffers from high replication factors. The $\mathcal{R}$ of Hybrid-BL is lower than PowerLyra on most graphs except *tw* and *wk*. The $\mathcal{R}$ of FSM depends on the choice of the splitting factor k. The smaller the value of k, the fewer subgraphs are split, which usually leads to a smaller $\mathcal{R}$. In most graphs, FSM achieves a $\mathcal{R}$ close to NE's under $p = 32$. Streaming partitioners have a shortcoming in terms of $\mathcal{R}$ compared to others, with DBH having the largest $\mathcal{R}$ followed by EBV and HDRF having the smallest $\mathcal{R}$. They also have a significant gap in $\mathcal{R}$ compared to other in-memory and hybrid partitioners.

5.2.3 Coefficient Variation of Vertex and Edge. The coefficient variation of vertex and edge are denoted as σ_V and σ_E, respectively. They indicate the discreteness of vertex and edge number among partitions, reflecting the degree of heterogeneity in ***memory overhead and computation workload*** during distributed tasks. In a homogeneous cluster, where the computing capabilities and communication capabilities between nodes are identical, the computing loads and memory overheads between machines should be as close as possible during distributed tasks. As indicated in Table 4, streaming partitioners usually have an advantage in terms of both σ_V and σ_E. DBH and PowerLyra exhibit the most outstanding σ_V because they both use hash for partitioning. EBV and HDRF follow closely behind. In-memory partitioners such as NE and HEP can only guarantee a low σ_E, but not σ_V. Hybrid-BL considers the fusion of multi-hop neighbors, which makes it impossible to obtain low σ on graphs with large degree differences. The dual-balanced partitioner BPart needs to optimize the balance of vertices and edges simultaneously, but it exhibits inferior performance in dual balance. MDBGP fails to complete the partitioning within 72 hours on all 11 graphs. FSMs achieve rather low σ_V on most graphs and the effect becomes more significant as k increases.

5.2.4 Time and Memory Overhead. Due to the space limit, we present the time and memory overhead on the three largest graphs *uu*, *u7*, and *sk*. The results are shown in Table 3, where 'MLE' and 'TLE' represent 'Memory Limit Exceeded' and 'Time Limit Exceeded' (the maximum partitioning time is set to 72 hours), respectively. Among the partitioners, MDBGP (40-thread partitioning) takes more time than other methods due to its involvement in a $O(n^2)$ intersection point operations. In contrast, the streaming algorithms demonstrate the fastest partitioning time. In terms of memory overhead, the streaming algorithms PowerLyra, DBH, and HDRF exhibit significant advantages, while the adjacency list-based algorithms such as BPart, NE, and Hybrid-BL require more memory. EBV sorts the edges according to the degree, which results in higher memory and time overhead compared to DBH and HDRF. The hybrid algorithms (HEPs) show intermediate performance in both time and memory overhead. The memory and time overheads of FSMs depend on the algorithms chosen for the Split Phase. From Table 3, it can be observed that FSMs roughly approximate the employed algorithms in terms of both time and memory.

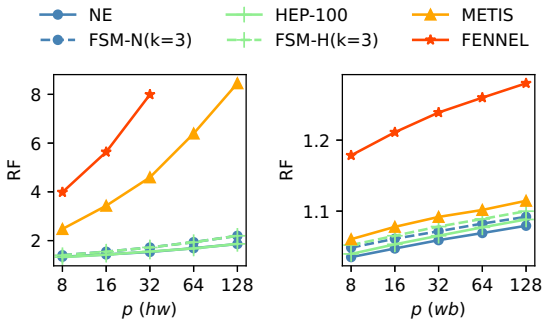


Figure 7: The trend of RF for different partitioners as p increases. FSM-N($k = 3$) and FSM-H($k = 3$) represent the FSM using NE and HEP, respectively, for partitioning with $k = 3$.

Table 5: Effects of α and β on the partitioning quality.

α	1.5	1.10	1.05	1.03
k	3	5	7	8
$\mathcal{B}_V$	1.32	1.10	1.05	1.02
$\mathcal{B}_E$	1.00	1.00	1.00	1.00
$\mathcal{R}$	1.11	1.12	1.12	1.13

β	1.35	1.1	1.05	1.03
k	5	5	6	6
$\mathcal{B}_V$	1.02	1.02	1.01	1.02
$\mathcal{B}_E$	1.35	1.10	1.05	1.03
$\mathcal{R}$	1.11	1.11	1.12	1.12

5.3 Effect of Parameters

5.3.1 Effect of the Number of Partitions p. In this study, we apply four state-of-the-art partitioners (NE, HEP, METIS, and FENNEL) to partition two graphs, *wb* and *hw*, with p ranging from 8 to 128. As shown in Figure 7, it is evident that all the values of $\mathcal{R}$ increase with the increase in p. This can be attributed to the fact that as the number of partitions p increases, there is a corresponding increase in the number of vertices at the partition boundaries. Consequently, this leads to unavoidable increases in the value of $\mathcal{R}$.

When analyzing the trends of the two graphs separately, we observe that increasing the value of p results in a significant rise in $\mathcal{R}$ for the *hw* graph. Particularly, for METIS, at $p = 128$, its $\mathcal{R}$ has already reached around four times that of $p = 8$. In contrast, the $\mathcal{R}$ for the *wb* graph remains consistently close to 1.0 across the entire range of p. For two FSM partitioners, we can observe that as p increases, FSM-N($k = 3$) and FSM-H($k = 3$) consistently achieve $\mathcal{R}$ that are similar to NE and HEP.

5.3.2 Effect of the Splitting Factor k. Fast merge prioritizes time efficiency, while precise merge often achieves better partition quality. In this study, we evaluate the effects of these two merge algorithms, using HEP-100[1] as the partitioner in the *Fine-grained Split* phase with $p = 32$ and $k = 2, 3, 4, 5, 6$. We evaluate them based on merging quality (Ω_V and $\mathcal{R}$) and merging time. The results on graphs *sk* and *tw* are presented in Figure 8.

When using the naive HEP-100 for partitioning, it achieves a value of Ω_V of 1.73 and 2.92 on *sk* and *tw*, respectively. With the FSM framework, the $\mathcal{R}$ and Ω_V obtained from the *Precise Merge* are usually lower than those from the *Fast Merge*. We observe that as k increases, FSM consistently maintains $\mathcal{R}$ and Ω_V at similar levels, resulting in a favorable $\mathcal{B}_V$. In particular, the *Precise Merge*

[1]NE cannot guarantee edge balance when p is large.

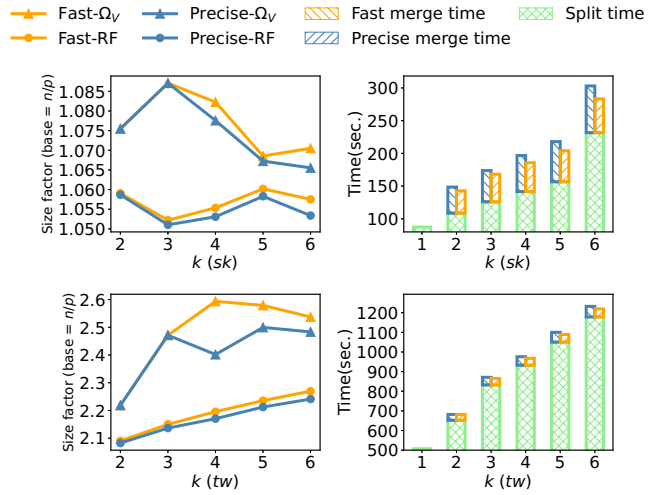


Figure 8: The merge effect and time of FSM-H v.s. k, where $p = 32$ and $k = 1$ indicates partitioned by naive HEP-100.

algorithm achieves an average $\mathcal{B}_V$ of 1.02 and 1.12 on the graphs *sk* and *tw*, respectively. For the *sk* graph (other similar graphs like *id*, *it*, *ar*, and *wb*), we find that FSM achieves a ***triple one*** score, where all $\mathcal{B}_E$, $\mathcal{B}_V$, and $\mathcal{R}$ are close to 1. This is close to the theoretical optimum, as any partition must satisfy $\mathcal{R} > 1$ when $p > 1$. The time gap between the two merge algorithms increases with k. However, even for a graph like *sk* with 51M vertices and 1.9B edges, the *Precise Merge* with $k = 5$ only takes 60 seconds.

5.3.3 Effect of α and β. We discuss how the dual-balanced constraints, α and β, affect the quality of FSM partitioning. To study the effect of α, we vary α from 1.03 to 1.50 by fixing $\beta = 1$. Table 5 presents the results on graph *uk*. It shows that stricter vertex balance requirements (i.e., smaller α) necessitate a larger k, at the same time it also results in higher replication factors $\mathcal{R}$. To study the effect of β, we vary β from 1.03 to 1.35 by fixing $\alpha = 1.03$. Compared to the case of fixing $\beta = 1$, relaxing the constraint on β makes it easier to achieve the target of $\alpha = 1.03$. It is worth noting that FSM maintains a lower replication factor $\mathcal{R}$ as well.

5.4 Results of Distributed Tasks

5.4.1 Accelerating Distributed Tasks. To evaluate the performance of FSM on downstream tasks, we conduct PageRank (PR), connected components (CC), and approximate diameter (AD) on a cluster of $p = 8$ machines. PR is fixed to 100 iterations to force all vertices to be active so that we can test the effect under heavy communication overhead. CC is implemented based on label propagation, with fewer active vertices in each iteration. AD estimates the diameter by computing reached vertex pairs for each hop [18], with all vertices being re-activated in each hop. We fix the number of hops to 10 due to its long elapsed time. We repeat each experiment 3 times and take the average as the result, ensuring that the standard error of the experiment time is less than 5%. For tasks that run out of memory, we label them as "MLE". For tasks that cannot be finished within 12 hours, we abort them and label them as "TLE". Tables 6, 7, and 8 list the results, where "-" indicates that the partitioning cannot be

Table 6: Runtime of distributed PageRank (sec). The lowest runtime is highlighted in bold, second lowest is underlined.

Graph	DBH	MDBGP	BPart	Hybrid-BL	PowerLyra	EBV	CLUGP	HDRF	2PS	HEP-1	HEP-10	HEP-100	NE	FSM-N(k=2)	FSM-N(k=3)	FSM-H(k=2)	FSM-H(k=3)
ar	329.23	361.90	255.27	226.50	217.80	222.30	235.13	216.23	173.60	179.47	180.50	176.00	192.70	166.53	**159.47**	164.33	173.37
hw	124.37	111.43	108.53	99.67	87.80	88.30	102.13	90.27	85.93	79.37	78.50	73.63	74.87	72.07	71.40	**68.17**	70.43
id	167.77	139.90	121.40	105.57	94.27	104.67	104.70	105.30	99.03	97.77	98.53	95.43	101.40	**92.83**	97.87	93.27	93.60
it	622.07	-	476.33	390.33	383.30	381.67	371.17	290.50	267.77	309.73	333.17	291.90	376.77	282.47	264.97	**249.97**	265.77
sk	767.60	-	828.17	477.60	523.47	510.20	412.47	436.27	373.60	334.30	337.63	323.10	341.60	332.67	311.50	308.17	**308.10**
tw	535.63	-	1028.97	MLE	424.90	479.10	843.23	438.57	494.80	422.40	405.17	394.40	395.80	423.13	420.50	**392.93**	398.83
u7	1527.50	-	1173.73	893.07	871.33	990.80	759.27	732.53	496.13	527.70	567.93	595.07	739.93	550.30	534.70	462.67	**449.37**
uk	648.80	-	448.30	368.53	421.50	377.97	314.10	315.60	281.83	307.20	294.83	294.33	321.50	277.60	261.90	274.53	**259.63**
uu	1892.30	-	1080.83	1371.20	MLE	1398.83	-	829.63	649.93	691.83	781.97	782.20	895.90	685.37	664.90	631.90	**606.83**
wb	1609.17	-	914.70	1121.27	866.33	921.30	872.10	853.97	733.93	836.27	765.97	748.83	782.23	778.20	741.77	**699.17**	699.30
wk	305.43	522.57	416.97	405.00	196.97	261.53	352.53	247.13	240.40	227.47	**192.60**	193.37	200.67	199.60	198.60	198.97	197.53

Table 7: Runtime of connected components (sec). The lowest runtime is highlighted in bold, second lowest is underlined.

Graph	DBH	MDBGP	BPart	Hybrid-BL	PowerLyra	EBV	CLUGP	HDRF	2PS	HEP-1	HEP-10	HEP-100	NE	FSM-N(k=2)	FSM-N(k=3)	FSM-H(k=2)	FSM-H(k=3)
ar	46.77	58.30	44.23	40.30	38.53	41.93	43.27	35.97	34.00	32.07	33.60	33.23	37.33	31.13	29.53	**28.67**	33.03
hw	12.73	14.03	14.17	12.40	10.37	10.90	12.00	10.30	9.87	9.57	9.20	9.17	8.47	9.37	8.33	8.83	**7.70**
id	24.30	25.87	20.87	20.07	**16.80**	21.37	19.97	17.27	18.23	18.50	18.57	19.47	20.70	17.97	17.33	17.77	16.87
it	77.90	-	80.07	75.47	62.97	65.70	66.30	50.37	55.87	49.07	51.57	52.87	60.87	49.90	48.43	46.17	**43.50**
sk	76.80	-	90.00	71.33	81.63	70.90	77.03	53.53	60.70	47.33	**44.00**	45.17	53.10	46.90	47.37	45.03	45.00
tw	64.13	-	103.67	MLE	63.73	66.43	100.97	61.23	77.03	57.27	56.80	55.47	61.83	56.33	55.83	**50.57**	51.63
u7	173.50	-	179.20	154.43	157.73	169.50	147.77	125.87	114.40	113.27	118.87	113.73	123.10	110.47	110.90	110.20	**108.77**
uk	104.90	-	83.60	82.23	86.23	83.23	71.63	61.80	64.20	57.47	67.37	66.10	73.20	57.57	**54.70**	57.50	61.20
uu	294.13	-	232.13	269.50	MLE	277.43	-	190.90	197.20	186.60	192.93	182.23	199.00	180.97	180.27	**174.67**	174.93
wb	509.60	-	444.87	473.73	423.10	480.37	444.03	430.77	409.97	399.53	432.57	421.30	432.13	412.83	**396.10**	415.50	418.57
wk	49.53	74.57	65.87	64.47	**34.43**	45.67	50.47	44.07	42.60	43.13	40.47	42.03	41.13	37.40	36.80	37.17	41.53

Table 8: Runtime of approximate diameter (sec). The lowest runtime is highlighted in bold, second lowest is underlined.

Graph	DBH	MDBGP	BPart	Hybrid-BL	PowerLyra	EBV	CLUGP	HDRF	2PS	HEP-1	HEP-10	HEP-100	NE	FSM-N(k=2)	FSM-N(k=3)	FSM-H(k=2)	FSM-H(k=3)
ar	954.67	1189.33	779.67	759.00	910.82	725.00	902.00	651.67	575.67	535.67	528.67	531.67	586.00	470.33	**439.00**	452.67	457.00
hw	233.00	292.00	200.00	225.00	239.91	149.67	211.33	133.33	162.00	130.00	144.67	127.67	131.33	112.33	**111.33**	116.33	115.33
id	304.00	477.00	229.00	244.00	249.54	284.00	189.00	200.67	199.33	167.00	185.33	210.00	339.67	153.00	146.00	**144.00**	146.00
it	1909.33	-	1543.00	1589.00	1705.89	1417.33	1684	1050.00	1042.00	1006.33	1073.33	1065.67	1376.67	954.67	861.33	**851.67**	867.67
sk	3243.33	-	2792.67	2423.00	2713.50	2131.67	2550.33	1859.33	1540.00	1393.67	1316.67	1333.33	1717.00	1414.67	1261.33	**1256.67**	1281.33
tw	2027.00	-	4324.33	MLE	1963.02	1900.67	4165.67	1713.00	2546.67	1634.67	1633.33	1816.67	2077.67	1611.33	1713.33	**1487.33**	1569.67
u7	MLE	-	MLE	MLE	MLE	MLE	MLE	MLE	MLE	MLE	MLE	MLE	MLE	MLE	MLE	TLE	**4326.00**
uk	1835.00	-	1185.67	1395.33	1940.11	1444.67	1145.67	946.67	980.67	918.00	1103.33	1157.33	1174.33	846.33	**753.33**	858.67	771.67
uu	MLE	-	MLE	MLE	MLE	MLE	-	MLE	MLE	MLE	MLE	MLE	MLE	MLE	MLE	MLE	MLE
wb	MLE	-	MLE	MLE	MLE	MLE	MLE	MLE	MLE	MLE	MLE	MLE	MLE	MLE	4587.33	4542.33	**2041.33**
wk	964.67	2363.33	1759.33	2286.33	702.78	918.67	1495.33	878.00	859.00	792.00	757.67	734.33	754.00	**700.00**	714.33	710.67	727.33

finished within 72 hours or exceeds the memory limit during the partitioning process.

Overall elapsed time. We conclude that using FSM-N($k = 3$) or FSM-H($k = 2$) generally results in the shortest elapsed time on most graph tasks. Although NE achieves the lowest $\mathcal{R}$, its time efficiency is generally not as good as HEP and even falls behind the streaming partitioner HDRF (on the *it* graph). HEP demonstrates excellent performance in PR and CC tasks, with overall performance second only to FSM. Among the streaming partitioners, HDRF exhibits the best performance, followed by EBV, and DBH performs the worst. The two re-streaming algorithms show impressive performance on graphs like *it*. This is likely due to the presence of clustering in graphs, which allows these algorithms to maintain good balance while achieving low $\mathcal{R}$. PowerLyra demonstrates good performance on graphs such as *tw* and *wk*, while Hybrid-BL performs better on graph *wb*.

We note that the improvement of FSM-N over NE is impressive. The maximum improvement on PR and CC tasks is 29.7% and 25.3%, respectively. Similar improvement is also observed in FSM-H.

Paging Fault due to vertex imbalance. In distributed environments with limited memory, large graph processing tasks often experience frequent page faults. In some cases, although the graph processing task does not exceed memory capacity, the memory usage approaches 100%, increasing the likelihood of page faults. These page faults contribute to higher CPU utilization by the `kswapd0` process, which is used for swapping pages. For instance, the AD tasks on graph *wb* partitioned by FSM-N(k=3) and FSM-H(k=2) are adversely affected by increasing page faults and additional CPU utilization, resulting in a long elapsed time. Hence, in distributed tasks,

the memory overhead on the largest partition should be minimized, even though it may not exceed memory.

5.4.2 Guidance on Applicability. Based on extensive experimental results on various datasets, we have summarized the following scenarios where the FSM framework is applicable: (I) Graphs with rich communities and high-degree vertices, such as *wb*, *it*, *uu*, and *id*. These communities and high-degree vertices can facilitate the fine-grained splitting phase of FSM, providing small replication factors even for a large number of partitions. (II) Downstream tasks that require high memory usage. Tasks such as triangle counting and approximate diameter often involve complex data structures that consume a significant amount of memory. This can lead to memory bottlenecks on individual machines. (III) When the cluster size is small, each machine needs to handle a large computational load, and the load imbalance in terms of computation and memory becomes more prominent. Using FSM to balance the load can significantly enhance the performance of tasks on small clusters.

6 RELATED WORK

In-memory partitioners. One of the most established categories of in-memory partitioning algorithms is multilevel partitioning algorithms, such as METIS [19], KaFFPa [33], and Scotch [30]. These algorithms emphasize cut size and vertex balance but tend to perform poorly in terms of edge balance. NE [39] is a typical edge partitioning algorithm that achieves excellent $\mathcal{R}$ by prioritizing the expansion of vertices with fewer external connections. HEP (along with NE++) [26] and DNE [15] adopt the NE idea and primarily focus on optimizing partition efficiency and scalability but often neglect the issue of vertex balance.

Streaming partitioners. Streaming algorithms [35] can be further classified into stateful and stateless categories. Stateful algorithms typically evaluate vertices or edges using specific scoring functions and allocate them to partitions with the highest scores. HDRF [31] prioritizes cutting high-degree vertices and performs well on power-law graphs. However, the scoring function does not consider vertex balance. FENNEL [36] is a vertex partitioning algorithm that takes into account both the number of adjacent and non-adjacent vertices. However, it does not consider edge balance. 2PS [27] and CLUGP [21] are two re-streaming algorithms. They utilize clustering algorithms to proactively gather global information, which aids in subsequent re-partitioning. The difference lies in the clustering strategy used. These re-streaming algorithms tend to yield better partitioning results but often sacrifice balance.

Grid [17] is a hashing-based edge partitioning algorithm that takes into account the load balancing of partitions. DBH [38] is a degree-based hashing edge partitioning algorithm, where the hashing function incorporates the idea of "high-degree vertices are preferentially replicated". Random partitioning [35] randomly assigns edges or vertices to partitions.

Dual-balanced partitioners. Zhang et al. propose the edge partitioning algorithm EBV [40], which incorporates both vertex and edge load as well as vertex replication into the scoring function and thus achieves a good dual balance. However, like other streaming algorithms, it can only achieve sub-optimal $\mathcal{R}$. Lin et al. extend the FENNEL algorithm and introduce a vertex partitioning scheme called BPart [24]. It achieves dual balance through multiple rounds of re-partitioning and combining. However, it requires simultaneous optimization of both vertex and edge balance during the combining phase and only considers extreme merges (merging the current maximum with the minimum), without guaranteeing an approximation ratio. In terms of $\mathcal{R}$, BPart incurs significant losses compared to FENNEL, while FENNEL is inferior to METIS in $\mathcal{R}$. Different from the discrete algorithms, MDBGP [1] uses Projected Gradient Descent to balance vertex-edge load. It exhibits high parallelism and emphasizes multi-dimension balance. However, it suffers from significant complexity and falls short in cut size, performing even worse than FENNEL.

Different from these methods, FSM employs an effective two-phase method that can benefit from any SOTA single-balanced partitioners and achieve remarkable replication factor.

Postprocessing algorithms. ParE2H and ParV2H [9] are task-specific algorithms that refine given partitions to shorten the elapsed time for specific tasks. However, they usually require higher training costs and are limited to optimizing a particular task. In contrast, FSM achieves fast partitioning and provides acceleration benefits for a wide range of tasks. Additionally, FSM generates high-quality partitions with dual balance. LS [14] adjusts edges and blocks of given partitions to achieve high-quality partitions under edge balance. However, similar to other edge partitioners, LS primarily focuses on reducing $\mathcal{R}$, which can potentially lead to a deterioration in vertex balance during the adjustment process.

Dynamic partitioners. Dynamic partitioners are used for partitioning dynamic graphs. They typically achieve dynamic optimization by migrating vertices and edges. Based on two optimization objectives: structural changes and computational load changes, dynamic partitioners can be divided into two categories. The former focuses on optimizing the dynamic graph structure, e.g., Leopard [16], Hermes [28], and Planar [41]. The latter considers the workload of the graph computation, e.g., Mizan [20] and CatchW [34].

FSM can be also extended to dynamic scenarios. For dynamically updated graphs, a buffer can be set up to cache the incoming updates. The buffer can then be partitioned using a single-balanced partitioner, and the resulting partitions can be merged with the existing bins using the Merge operation. This allows for the incorporation of dynamic updates while maintaining the dual balance.

7 CONCLUSION

In this paper, we introduce a lightweight and efficient partitioning framework FSM that consists of two phases, namely *fine-grained splitting* and *subgraph merging*. The experimental results on large-scale graphs have demonstrated that the proposed framework FSM achieves a remarkable replication factor while achieving both vertex and edge balances, outperforming state-of-the-art partitioners. Furthermore, the performance of downstream tasks can be significantly improved benefiting from the partitioning result.

ACKNOWLEDGMENTS

This work was supported by National Natural Science Foundation of China (Grant No. U23A20496, 61932001, and U20A20174), Huawei Technologies Co., Ltd. (Grant No. TC20220804023). Weiguo Zheng is the corresponding author.

REFERENCES

[1] Dmitrii Avdiukhin, Sergey Pupyrev, and Grigory Yaroslavtsev. 2019. Multi-Dimensional Balanced Graph Partitioning via Projected Gradient Descent. *Proc. VLDB Endow.* 12, 8 (apr 2019), 906–919. https://doi.org/10.14778/3324301.3324307

[2] Paolo Boldi, Andrea Marino, Massimo Santini, and Sebastiano Vigna. 2014. BUbiNG: massive crawling for the masses. In *23rd International World Wide Web Conference, WWW '14, Seoul, Republic of Korea, April 7-11, 2014, Companion Volume*, Chin-Wan Chung, Andrei Z. Broder, Kyuseok Shim, and Torsten Suel (Eds.). ACM, 227–228. https://doi.org/10.1145/2567948.2577304

[3] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011*, Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar (Eds.). ACM, 587–596. https://doi.org/10.1145/1963405.1963488

[4] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM Press, Manhattan, USA, 595–601.

[5] Florian Bourse, Marc Lelarge, and Milan Vojnovic. 2014. Balanced Graph Edge Partition. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, New York, USA) *(KDD '14)*. Association for Computing Machinery, New York, NY, USA, 1456–1465. https://doi.org/10.1145/2623330.2623660

[6] Coen Bron and Joep Kerbosch. 1973. Algorithm 457: Finding All Cliques of an Undirected Graph. *Commun. ACM* 16, 9 (sep 1973), 575–577. https://doi.org/10.1145/362342.362367

[7] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. PowerLyra: differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 1, 15 pages. https://doi.org/10.1145/2741948.2741970

[8] James Cheng, Yiping Ke, Ada Wai-Chee Fu, Jeffrey Xu Yu, and Linhong Zhu. 2011. Finding Maximal Cliques in Massive Networks. *ACM Trans. Database Syst.* 36, 4, Article 21 (dec 2011), 34 pages. https://doi.org/10.1145/2043652.2043654

[9] Wenfei Fan, Ruiqi Xu, Qiang Yin, Wenyuan Yu, and Jingren Zhou. 2022. Application-Driven Graph Partitioning. *The VLDB Journal* 32, 1 (apr 2022), 149–172. https://doi.org/10.1007/s00778-022-00736-2

[10] Michael R. Garey and David S. Johnson. 1990. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA.

[11] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*. USENIX Association, 17–30.

[12] 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 *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 599–613.

[13] R. L. Graham. 1969. Bounds on Multiprocessing Timing Anomalies. *SIAM J. Appl. Math.* 17, 2 (1969), 416–429. https://doi.org/10.1137/0117039 arXiv:https://doi.org/10.1137/0117039

[14] Zhenyu Guo, Mingyu Xiao, Yi Zhou, Dongxiang Zhang, and Kian-Lee Tan. 2021. Enhancing Balanced Graph Edge Partition with Effective Local Search. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*. AAAI Press, 12336–12343. https://doi.org/10.1609/AAAI.V35I14.17464

[15] Masatoshi Hanai, Toyotaro Suzumura, Wen Jun Tan, Elvis Liu, Georgios Theodoropoulos, and Wentong Cai. 2019. Distributed Edge Partitioning for Trillion-Edge Graphs. *Proc. VLDB Endow.* 12, 13 (sep 2019), 2379–2392. https://doi.org/10.14778/3358701.3358706

[16] Jiewen Huang and Daniel J. Abadi. 2016. Leopard: lightweight edge-oriented partitioning and replication for dynamic graphs. *Proc. VLDB Endow.* 9, 7 (mar 2016), 540–551. https://doi.org/10.14778/2904483.2904486

[17] Nilesh Jain, Guangdeng Liao, and Theodore L. Willke. 2013. GraphBuilder: Scalable Graph ETL Framework. In *First International Workshop on Graph Data Management Experiences and Systems* (New York, New York) *(GRADES '13)*. Association for Computing Machinery, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/2484425.2484429

[18] U. Kang, Charalampos Tsourakakis, Ana Paula Appel, Christos Faloutsos, and Jure Leskovec. 2008. HADI: Fast Diameter Estimation and Mining in Massive Graphs with Hadoop. *Journal Acm Transactions on Knowledge Discovery from Data* (2008).

[19] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.* 20, 1 (dec 1998), 359–392.

[20] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. 2013. Mizan: A System for Dynamic Load Balancing in Large-Scale Graph Processing. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic) *(EuroSys '13)*. Association for Computing Machinery, New York, NY, USA, 169–182. https://doi.org/10.1145/2465351.2465369

[21] Deyu Kong, Xike Xie, and Zhuoxu Zhang. 2022. Clustering-based Partitioning for Large Web Graphs. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 593–606. https://doi.org/10.1109/ICDE53745.2022.00049

[22] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[23] Dongsheng Li, Yiming Zhang, Jinyan Wang, and Kian-Lee Tan. 2019. TopoX: topology refactorization for efficient graph partitioning and processing. *Proc. VLDB Endow.* 12, 8 (apr 2019), 891–905. https://doi.org/10.14778/3324301.3324306

[24] Shuai Lin, Rui Wang, Yongkun Li, Yinlong Xu, John C.S. Lui, Fei Chen, Pengcheng Wang, and Lei Han. 2023. Towards Fast Large-Scale Graph Analysis via Two-Dimensional Balanced Partitioning. In *Proceedings of the 51st International Conference on Parallel Processing* (Bordeaux, France) *(ICPP '22)*. Association for Computing Machinery, New York, NY, USA, Article 37, 11 pages. https://doi.org/10.1145/3545008.3545060

[25] Grzegorz Malewicz, Matthew H. Austern, Aart J.C 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* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 135–146. https://doi.org/10.1145/1807167.1807184

[26] Ruben Mayer and Hans-Arno Jacobsen. 2021. Hybrid Edge Partitioner: Partitioning Large Power-Law Graphs under Memory Constraints. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1289–1302. https://doi.org/10.1145/3448016.3457300

[27] Ruben Mayer, Kamil Orujzade, and Hans-Arno Jacobsen. 2022. Out-of-Core Edge Partitioning at Linear Run-Time. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 2629–2642. https://doi.org/10.1109/ICDE53745.2022.00242

[28] Daniel Nicoara, Shahin Kamali Khuzaima Daudjee, and Lei Chen. 2015. Hermes: Dynamic Partitioning for Distributed Social Network Graph Databases. (2015).

[29] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1998. The PageRank Citation Ranking: Bringing Order to the Web. *Stanford Digital Libraries Working Paper* (1998).

[30] François Pellegrini and Jean Roman. 1996. SCOTCH: A Software Package for Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. In *Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking (HPCN Europe 1996)*. Springer-Verlag, Berlin, Heidelberg, 493–498.

[31] Fabio Petroni, Leonardo Querzoni, Khuzaima Daudjee, Shahin Kamali, and Giorgio Iacoboni. 2015. HDRF: Stream-Based Partitioning for Power-Law Graphs. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management* (Melbourne, Australia) *(CIKM '15)*. Association for Computing Machinery, New York, NY, USA, 243–252. https://doi.org/10.1145/2806416.2806424

[32] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. https://networkrepository.com

[33] Peter Sanders and Christian Schulz. 2011. Engineering Multilevel Graph Partitioning Algorithms. In *Proceedings of the 19th European Conference on Algorithms* (Saarbrücken, Germany) *(ESA'11)*. Springer-Verlag, Berlin, Heidelberg, 469–480.

[34] Zechao Shang and Jeffrey Xu Yu. 2013. Catch the Wind: Graph workload balancing on cloud. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 553–564. https://doi.org/10.1109/ICDE.2013.6544855

[35] Isabelle Stanton and Gabriel Kliot. 2012. Streaming Graph Partitioning for Large Distributed Graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Beijing, China) *(KDD '12)*. Association for Computing Machinery, New York, NY, USA, 1222–1230. https://doi.org/10.1145/2339530.2339722

[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] Shiv Verma, Luke M. Leslie, Yosub Shin, and Indranil Gupta. 2017. An experimental comparison of partitioning strategies in distributed graph processing. *Proceedings of the Vldb Endowment* 10, 5 (2017), 493–504.

[38] Cong Xie, Ling Yan, Wu-Jun Li, and Zhihua Zhang. 2014. Distributed Power-law Graph Computing: Theoretical and Empirical Analysis. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*. 1673–1681. https://proceedings.neurips.cc/paper/2014/hash/67d16d00201083a2b118dd5128dd6f59-Abstract.html

[39] 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* (Halifax, NS, Canada) *(KDD '17)*. Association for Computing Machinery, New York, NY, USA, 605–614. https://doi.org/10.1145/3097983.3098033

[40] Shuai Zhang, Zite Jiang, Xingzhong Hou, Zhen Guan, Mengting Yuan, and Haihang You. 2021. An Efficient and Balanced Graph Partition Algorithm for the Subgraph-Centric Programming Model on Large-scale Power-law Graphs. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. 68–78. https://doi.org/10.1109/ICDCS51616.2021.00016

[41] Angen Zheng, Alexandros Labrinidis, and Panos K. Chrysanthis. 2016. Planar: Parallel lightweight architecture-aware adaptive graph repartitioning. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. 121–132. https://doi.org/10.1109/ICDE.2016.7498234