applied sciences

MDPI

Article

A Star-Based Graph Partitioning Approach to Support "Master–Slave" Parallel Computing

Guosun Zeng [1,2], Tengteng Cheng [1,2,*] and Yufei Jin [1,2]

[1] School of Computer Science and Technology, Tongji University, Shanghai 201804, China; gszeng@tongji.edu.cn (G.Z.); jinyufei@tongji.edu.cn (Y.J.)
[2] National Engineering & Technology Center of High Performance Computer, School of Computer Science and Technology, Tongji University, Shanghai 201804, China
* Correspondence: chengtengteng2080@gmail.com

Abstract

Graph partitioning is a key step in the parallel and distributed processing of big graphs, aimed at improving efficiency while balancing load and reducing communication overhead. However, conventional partitioning methods often produce irregular inter-block communication patterns, leading to complex coordination costs. To address this issue, we propose a star-based graph partitioning approach that organizes partition blocks into a master–slave topology, where all inter-block communication is centralized through a master block. This structured design simplifies communication patterns and enhances parallel execution efficiency. Based on a cost-aware representation of graph processing, we formulate a partitioning objective that minimizes the overhead of the master block while maintaining balanced slave partitions. To achieve this, we develop a two-stage framework including initial partition generation and iterative refinement. The refinement stage incorporates a master minimization strategy and a slave balancing strategy to further improve partition quality. Extensive experiments on multiple benchmark datasets demonstrate that the proposed method effectively reduces communication overhead, improves parallelism, and provides an interpretable partitioning structure for distributed graph processing.

Keywords: graph partitioning; master–slave graph; master block minimization; slave block balancing

1. Introduction

In the current information era, graphs are becoming larger and larger, and traditional single-machine systems can no longer efficiently process such big graph data. Consequently, parallel and distributed computing has become a dominant paradigm for processing these graphs. In such systems, graph partitioning is the first and critical step, as the quality of partitioning directly affects computational efficiency. Existing graph partitioning methods, such as Ja-Be-Ja [1], LDG [2], and Metis [3,4], generally follow a divide-and-conquer strategy by decomposing a large graph into multiple subgraphs that are distributed across computing nodes. In these methods, edges across different blocks (i.e., inter-block edges) are typically unavoidable and may exist between arbitrary pairs of blocks. Although modern distributed systems can handle such dependencies through message passing and synchronization, these irregular communication patterns may still introduce additional coordination overhead. From a structural perspective, a communication-efficient partitioning pattern is one in which inter-block dependencies are organized in a controlled manner, rather than being arbitrarily distributed. Ideally, if there are no inter-block edges among the target blocks,

Check for updates

Academic Editor: Pedro Couto

Received: 6 September 2026
Revised: 18 October 2026
Accepted: 20 October 2026
Published: 22 October 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

they can be executed in parallel. In practice, however, obtaining such a partitioning result is almost impossible because of the inherent structural dependencies in graphs. A feasible partitioning is one in which a single target block is connected to all other blocks, while the remaining blocks have no inter-block edges between them. Motivated by this observation, we consider a partitioning structure where one block serves as a central coordinator, and all other blocks only communicate with this central block while remaining independent of each other. In this configuration, inter-block communication is concentrated on a single block, forming a star-shaped topology. This structure is related to vertex-separator formulations such as the k-vertex cut problem, but the underlying motivation is different. Rather than focusing on graph separation itself, we aim to construct a partition structure that is compatible with the execution requirements of the master–slave computing model. Specifically, one block serves as the master for coordinating inter-block dependencies, while the remaining blocks act as mutually independent slaves.

This structured partition naturally corresponds to the master–slave computing model widely used in parallel and distributed systems. In this model, the master process is responsible for coordination tasks such as scheduling, synchronization, and aggregation, while slave processes execute subtasks independently without direct communication among themselves. This model has been adopted in graph processing systems such as Pregel [5], Giraph [6], and GraphX [7] due to its simplicity, scalability, and reduced communication complexity. However, existing graph partitioning methods, which are broadly optimized for load balance, communication efficiency, and locality, rarely produce partition results that explicitly align with such a structured communication pattern. Instead, they treat all blocks uniformly, without considering differentiated roles in communication and coordination. To bridge this gap, we propose a graph partitioning approach that explicitly constructs a master–slave topology, thereby enabling a more structured and efficient communication mechanism in distributed graph processing. Compared with conventional graph partitioning methods that generally treat partition blocks uniformly, the proposed partitioning approach explicitly incorporates the execution requirements of the master–slave computing model by organizing partition blocks into a star-shaped topology with one master and mutually independent slave blocks. Although this structure is related to the structural idea of k-vertex cut, the proposed approach focuses on graph partitioning for parallel computing rather than graph separation itself. The main contributions of this paper are summarized as follows:

(1) We propose a structured graph partitioning paradigm that organizes partition blocks into a master–slave topology, enforcing a star-shaped communication pattern.
(2) Based on this paradigm, we develop a two-stage partitioning framework, including an initial feasibility-guided partition generation and a refinement process with master minimization and slave balancing strategies.
(3) We design a partition evaluation metric tailored to the master–slave structure, capturing both centralized communication cost and load balance among slave blocks.

The remainder of this paper is organized as follows. After reviewing related work in Section 2, we introduce the fundamental concepts of graph partitioning and parallel graph processing, as well as the partitioning objectives, in Section 3. The proposed master–slave partitioning approach is described in Section 4. The experimental evaluation of the proposed approach is covered in Section 5, and the paper concludes with a summary in Section 6.

2. Related Work

Graph partitioning plays a fundamental role in parallel and distributed graph processing, with the primary objectives of balancing computational load, minimizing communication overhead, and improving intra-block computation efficiency. Existing partitioning methods can be broadly classified into approaches focused on load balancing, communication minimization, and locality optimization. Recent advances in large-scale graph processing systems and new application paradigms have further diversified these approaches, including multilevel and multi-granularity strategies [8,9], learning-based and adaptive partitioning [10], and frameworks for graph neural networks [11]. Despite these differences, they can still be understood within the three fundamental objectives of load balance, communication efficiency, and locality optimization.

Load-balancing partitioning methods. These approaches aim to distribute the computational load evenly across target blocks to prevent overloaded nodes. Early methods relied on simple metrics such as vertex or edge counts [1–4]. Building on these, several algorithms achieve load balance through iterative vertex exchange (Kernighan–Lin [12], Fiduccia–Mattheyses [13]), greedy vertex assignment (LDG [2]), or multilevel coarsening-refinement (Metis [3,4]). However, balancing only vertex or edge counts does not necessarily reflect true computational cost. To address this, subsequent work incorporates system-level heterogeneity: Zoltan [14] allocates partition sizes in proportion to processor performance; Zeng et al. [15] jointly consider processing power, bandwidth, and memory capacity; Shen et al. [16] exploit CPU/GPU differences; Zhang et al. [17] consider thread-level execution density and memory access locality.

Communication-minimization methods. These methods seek to reduce inter-block edges or limit vertex replication, thereby reducing communication and synchronization overhead. Representative approaches include PT-Scotch [18], which uses graph reordering and neighborhood search; FENNEL [19], which jointly considers load balance and connectivity; PowerLyra [20], which differentiates high- and low-degree vertices; and S5P [21], which exploits graph skewness to minimize replication. Beyond graph-level heuristics, recent work also considers network heterogeneity: Zhou et al. [22] incorporate bandwidth differences across datacenters; GrapH [23] designs traffic-aware vertex-cut strategies; Walshaw et al. [24] and Huang et al. [25] adopt hierarchical strategies to minimize communication latency in multi-node systems.

Locality-optimization partitioning approaches. These methods optimize block structures to improve intra-block computation efficiency by preserving structural autonomy and data locality. Community-detection-based approaches such as Girvan–Newman [26] and Louvain [27] group densely connected vertices into the same partition. Onizuka et al. [28] maximize modularity to improve data reuse. For application-specific scenarios, FlowCutter [29] generates low-cut, high-internal-connectivity subgraphs for road networks. Runtime-aware approaches such as GraphIt [30], PARAGON [31], ARGO [32], and Jafari et al. [33] adapt partitions to underlying system resources, including cache, thread contention, and data migration costs.

Existing methods have made considerable progress in balancing loads, reducing communication overhead, and improving intra-block execution efficiency. A common limitation, however, is that the target blocks often contain many inter-block edges, which can reduce the efficiency of distributed processing. To overcome this limitation, a partitioning strategy inspired by the k-vertex cut problem [34–38] is considered, which investigates how removing a small set of vertices can split a graph into disconnected components. This principle guides the creation of target blocks where slave blocks have no inter-block edges between them. Such a structure enables concurrent processing of all slave blocks and reduces synchronization overhead, thereby improving the efficiency of distributed graph processing.

3. Graph Partitioning and Graph Parallel Processing

3.1. Basic Concepts of Graph Partitioning

Let $G = (V, E, w, \omega)$ be a graph with the vertex set V, the edge set E, the vertex weight function $w(\cdot)$, and the edge weight function $\omega(\cdot)$. A k-way partition of G is defined as $P(G,k) = \{p_1, p_2, \ldots, p_k\}$, where each partition $p_i = (V_i, E_i)$ satisfies $V_1 \cup V_2 \cup \ldots \cup V_k = V$, $V_i \cap V_j = \emptyset$, $i \neq j \in \{1, 2, \ldots, k\}$. A vertex $v \in V_i$ is a boundary vertex of p_i if it has at least one neighbor $u \notin V_i$; the set of all boundary vertices of p_i is denoted by $B(p_i)$. The set of cut edges connecting partitions p_i and p_j is given by $CE_{ij} = \left\{(u,v) \middle| u \in V_i, v \in V_j\right\}$, the set of cut edges of partition p_i is $CE_i = \bigcup_{j=1}^{k} CE_{ij}$, and the set of all cut edges is $CE = \bigcup_{i=1}^{k} CE_i$.

3.2. Master–Slave Parallel Processing Model

The master–slave model is a classical and widely adopted paradigm in parallel and distributed computing, as shown in Figure 1. It has a star-shaped structure, consisting of a single master process and multiple slave processes. The master handles global coordination, task assignment, state monitoring, data transfer, and result aggregation, while each slave executes only its assigned subtask and communicates exclusively with the master; slaves do not communicate directly with each other. This communication pattern reduces complexity and allows all slaves to run tasks fully in parallel.

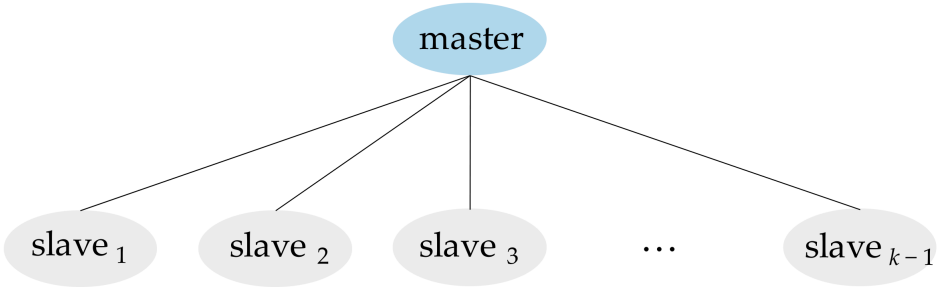


Figure 1. The master–slave parallel processing model.

Inter-block dependencies in parallel and distributed graph processing often require frequent synchronization and data exchange. By contrast, in the master–slave computing model illustrated in Figure 1, all communication occurs through a single master, allowing slave blocks to be processed concurrently, thereby reducing overall execution time. For example, if multiple computation blocks need to exchange information, organizing them in a master–slave structure ensures that each slave communicates only with the master, rather than with every other block. Topologically, this structure reduces the number of distinct inter-block communication pairs from $O(k^2)$ (in a fully connected mesh topology) to $O(k)$. We clarify that this reduction primarily describes the topology of potential communication relationships between partition blocks. It does not directly equate to a proportional reduction in the total volume of data exchanged or the actual runtime communication cost, which also depend on message size per synchronization step, communication frequency, synchronization protocol overhead, and underlying network properties such as latency and bandwidth contention.

3.3. Graph Application Algorithms

Graph data consist of entities (vertices) and the relationships (edges) connecting these entities. Efficient processing and analysis of graphs rely on specific application algorithms, which vary considerably in both functionality and computational behavior. Considering both the scope of graph data accessed and the dependence on additional input parameters, graph algorithms can be broadly classified into four categories. (1) Global iterative algorithms: These algorithms operate over the entire graph using only its structural information, typically updating all vertices or edges in each iteration. Representative algorithms include PageRank, GCN, Louvain, and DeepWalk, among others. (2) Local expansion algorithms: Also relying solely on graph structure, these algorithms focus each iteration on a subset of vertices or edges, expanding outward from a specific vertex or local region. Typical examples include BFS, DFS, Prim, and Kruskal. (3) Global constraint algorithms: These require additional input conditions or constraints and perform optimization or feasibility computations across the entire graph. Examples include GAT and GraphRNN. (4) Local expansion constraint algorithms: These combine input constraints with a limited computation scope, typically solving locally constrained optimization problems. Representative examples include Dijkstra's shortest-path algorithm, the Stoer–Wagner algorithm for computing s-t minimum cuts, and heuristic approaches for the traveling salesman problem. Some algorithms, such as VF2 (subgraph isomorphism) and Ford–Fulkerson (maximum flow), may involve complex subgraph construction or graph transformations. Despite their additional complexity, their data access patterns and task dependencies still align with the local expansion constraint algorithms category.

Global iterative algorithms are a class of structure-driven algorithms that are widely used in practice due to their ability to extract meaningful information from graph structures. These algorithms repeatedly process the entire graph until convergence, which can result in substantial computational costs. A key feature of these algorithms is their stable iterative logic and localized communication, where updates depend primarily on direct neighbors; consequently, they are often applied to undirected graphs. For example, PageRank and Louvain are widely applied to ranking web search results, analyzing social network influence, and detecting communities, while graph representation learning methods such as GCN, Node2Vec, and DeepWalk are used for recommendation, financial risk prediction, and predictive modeling in bioinformatics. The structure-driven nature of these algorithms highlights the value of partitioning a big graph into blocks with a master–slave structure. Such a partitioning approach not only matches the requirements of modern distributed graph processing platforms for parallelism and load balancing, but also provides a robust foundation for efficient and scalable analysis of big graph data.

3.4. Graph Processing Cost and Load Graph

Given a graph $G = (V, E, w, \omega)$ and an application algorithm A, suppose that G is divided into k blocks $p_1, p_2, \ldots, p_k$. The processing cost of p_i under algorithm A, denoted as $CA(p_i)$, consists of its computation and communication costs; i.e., $CA(p_i) = C^f_A(p_i) + C^g_A(p_i)$. The computational cost $C^f_A(p_i) = \sum_{v \in V_i} f_A(v)$ is the sum of costs over all vertices within p_i, and the communication cost $C^g_A(p_i) = \sum_{e \in CE_i} g_A(e)$ is the sum of costs over all cut edges incident to p_i, where $i = 1, 2, \ldots, k$, $f_A(v)$ is the computational cost of a single vertex v, while $g_A(e)$ quantifies the communication cost associated with edge e.

To explicitly characterize the execution costs of an algorithm A on a graph G, we introduce the concept of a load graph. A load graph is an augmented representation that combines the original graph topology with algorithm-specific execution properties, including vertex computation costs, vertex dependencies, and associated communication overheads. Formally, a load graph is defined as a quadruple $G_L = (V_L, E_L, w_L, \omega_L)$, where V_L is the set of vertices, and E_L represents the dependency relationship between vertices; $w_L : V_L \to \mathbb{R}^+$ assigns computation cost to each vertex, and $\omega_L : E_L \to \mathbb{R}^+$ assigns communication cost to each edge. In this work, we assume $V_L = V$ and $E_L = E$, while $w_L(v) = f_A(v)$ and $\omega_L(e) = g_A(e)$, for any $v \in V_L$ and $e \in E_L$. Thus, partitioning the load graph G_L is equivalent to partitioning the original graph G under cost-aware objectives. In practice, the cost functions $w_L(\cdot)$ and $\omega_L(\cdot)$ depend on both graph structural features (e.g., degree and clustering coefficient) and algorithm characteristics (e.g., computation logic, iteration patterns, and data access). To obtain realistic cost estimations, these functions can be learned from historical execution data using a polynomial regression-based model [39].

3.5. Partitioning Objectives and Formal Representation

For clarity in the subsequent discussion, we introduce a new type of graph, referred to as the master–slave graph, defined as follows:

Definition 1 (master–slave graph). *Let $G = (V, E, w, \omega)$ be a graph composed of k subgraphs $\{G_0^m, G_1^s, G_2^s, \ldots, G_{k-1}^s\}$, where $G_0^m = (V_0^m, E_0^m)$ is the master block, $G_i^s = (V_i^s, E_i^s)$ is the i-th slave block. The graph G is called a master–slave graph if its subgraphs satisfy the following conditions for all distinct $i, j \in \{1, 2, \ldots, k-1\}$: (1) $V_0^m \cup V_1^s \cup V_2^s \cup \ldots \cup V_{k-1}^s = V$; (2) $V_0^m \cap V_i^s = \emptyset$, $V_i^s \cap V_j^s = \emptyset$; (3) No edge $(u, v) \in E$ (u,v) exists with $u \in V_i^s$ and $v \in V_j^s$; (4) There exists at least one edge $(u, v) \in E$ such that $u \in V_0^m$ and $v \in V_i^s$.*

A master–slave graph exhibits a star topology: the master block is connected to all slave blocks, while no inter-block edges exist. To enable efficient parallel and distributed graph processing, we propose an approach for partitioning a graph into blocks that can form a master–slave graph, with the objectives of minimizing the master block and balancing the slave blocks. We refer to this approach as the master–slave graph partitioning. Formally, given a graph $G = (V, E, w, \omega)$ and an application algorithm A, the corresponding load graph is $G_L = (V, E, w_L, \omega_L)$. The partitioning task is to divide G into k blocks that can form a master–slave graph with a minimal master block and balanced slave blocks under algorithm A, which is equivalent to partitioning G_L into k blocks. It is denoted as $MSP(G, A, k) = MSP(G_L, k) = \{G_0^m, G_1^s, G_2^s, \ldots, G_{k-1}^s\}$, and the corresponding objective is:

$$\min_{G_0^m \in MSP(G_L,k)} \left(\sum_{v \in V_0^m} w_L(v) + \sum_{e \in CE_0} \omega_L(e) \right) \tag{1}$$

subject to:

$$(1 - \varepsilon)\overline{C}_A \le C_A(G_i^s) \le (1 + \varepsilon)\overline{C}_A \tag{2}$$

$$\bigcup_{i=1}^{k-1} (E_i^s \cup CE_i) \cup E_0^m = E \tag{3}$$

where $\overline{C}_A = \sum_{i=1}^{k-1} \left(\sum_{v \in V_i^s} f_A(v) + \sum_{e \in CE_i} g_A(e) \right) / (k-1)$ is the average computational cost of all slave blocks, $C_A(\cdot)$ denotes the processing cost of a partition block under algorithm A, CE_i represents the edge set between G_i^s and G_0^m, $0 < \varepsilon < 1$ is a balancing factor, and CE_0 is the set of edges between the master block and all slave blocks. Equation (2) enforces the load-balancing constraint on the slave blocks, while Equation (3) ensures that cut edges occur only between the master block and the slave blocks. These constraints distinguish the proposed partitioning problem from conventional graph partitioning and the k-vertex cut

problem in terms of partition structure and load requirements. Table 1 summarizes the key differences among the three formulations.

Table 1. Comparison of conventional graph partitioning, k-vertex cut, and the proposed partitioning problem.

	Traditional Graph Partitioning	k-Vertex Cut	Proposed Partitioning Problem
Partition Structure	k partition blocks with no specific role assignment	One small block is connected to at least $k-1$ other blocks, while the other blocks are mutually disconnected	One master block and $k-1$ slave blocks; slave blocks are mutually disconnected
Objective	Minimize the number of inter-block edges	Minimize the vertex set to separate the graph	Minimize the computational cost of the master block
Constraints	The k blocks are approximately balanced	Graph must be split into at least $k-1$ disconnected blocks	Slave blocks are computationally balanced

It is important to note that the existence of such a structured partition is not guaranteed for arbitrary graphs. Therefore, in the following section, we introduce a set of necessary conditions to characterize the feasibility of partitioning a graph into the desired form.

4. An Approach for Partitioning a Graph into Master–Slave Blocks

4.1. Necessary Conditions for Master–Slave Partitioning

Our partitioning strategy is inspired by the k-Vertex Cut Problem (k-VCP) [34]. In graph theory, the k-VCP addresses the task of removing a subset of vertices and their incident edges so that the resulting graph contains at least k disconnected components. Building upon this problem, our goal is to partition a graph into k blocks that can form a master–slave graph. Specifically, we aim to select a minimal set of vertices as the master block, while the remaining graph is divided into $k-1$ roughly balanced subgraphs serving as the slave blocks. This can be viewed as a specialized and constrained variant of the k-VCP, with additional structural and load-balancing requirements on the resulting partition. Since the k-VCP is NP-hard [36], these additional requirements make the master–slave partitioning problem computationally challenging. In real-world scenarios, a feasible partition that satisfies these requirements may not always exist. Hence, before pursuing detailed solution methods, it is essential to specify the necessary conditions for such a partition to be feasible.

A fundamental property of graph connectivity establishes the basis for the following lemma. It is well-known that a connected graph $G = (V, E)$ must contain a spanning tree with exactly $|V| - 1$ edges [40]. This implies that for a graph with at least k connected components, any acyclic subgraph has at most $|V| - k$ edges. Therefore, the k-VCP can be reformulated as the problem of finding a minimal subgraph $S \subset G$ whose removal results in at least k components.

Lemma 1. *Let $G = (V, E)$ be a graph; if a subgraph $S = (V_s, E_s) \subset G$ is a solution to the k-VCP, then any acyclic subgraph of $G \backslash S$ has at most $|V| - |V_s| - k$ edges.*

Lemma 1 captures a global structural requirement for the k-VCP. In practice, however, verifying this condition is computationally prohibitive and offers little insight into whether a k-VCP solution exists. To address these limitations, we focus on local structural properties to derive criteria that are easier to check and apply. Given a graph $G = (V, E)$ and a vertex set $V' \subset V$, let $N(V') = \bigcup_{v \in V'} N(v) \backslash V'$ denote the union of the neighborhoods of all vertices in V', excluding the vertices in V' itself. Based on this, we propose the following proposition as a necessary condition for a k-block master–slave partition.

Proposition 1. *A necessary condition for a graph to have a k-block master–slave partition is the existence of a nonempty vertex set V' such that the subgraph induced by $N(V')$ contains at least $k-1$ connected components.*

Proof of Proposition 1. Suppose a graph $G = (V, E)$ can be partitioned into k blocks with master–slave structure, i.e., $G_0^m, G_1^s, G_2^s, \ldots, G_{k-1}^s$, where $G_0^m = \left(V_0^m, E_0^m\right)$ is the master block, $G_i^s = (V_i^s, E_i^s)$ is the i-th slave block, $i = 1, 2, \ldots k-1$. Since each slave block is adjacent to the master block, $N\left(V_0^m\right)$ contains at least one vertex from each slave block. Moreover, no edges exist between distinct slave blocks. Therefore, vertices belonging to different slave blocks cannot be connected in the subgraph induced by $N\left(V_0^m\right)$. Consequently, this subgraph contains at least $k-1$ connected components. Hence, the stated condition is necessary for a k-block master–slave partition. □

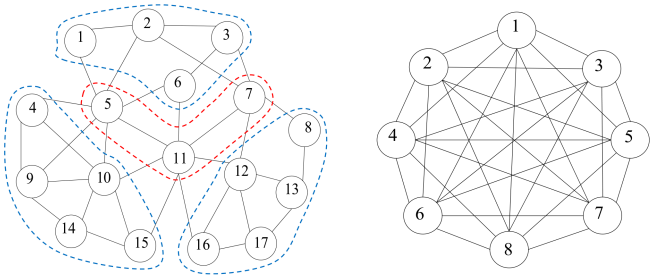
Proposition 1 provides a local structural condition that is necessary for a master–slave partition. Specifically, the neighborhood of the master block must contain at least $k-1$ connected components, corresponding to the $k-1$ slave blocks. This condition provides a basis for identifying candidate master blocks and assessing the feasibility of a master–slave partition. The feasibility of a master–slave partition is also constrained by the solvability of the k-VCP, since removing the master block leaves mutually disconnected slave blocks. Proposition 2 gives a structural condition under which the k-VCP has no solution.

Proposition 2. *If every subgraph induced by any k vertices in a graph is connected, then the graph has no solution to the k-VCP.*

Proof of Proposition 2. Consider a graph $G = (V, E)$ such that every subgraph induced by any k vertices is connected. Assume that a k-VCP solution exists. Let $S \subset G$ be a subgraph whose removal produces a graph $G' = G \backslash S$ with at least k disconnected components, denoted G_1, G_2, ..., G_k, with vertex sets V_1, V_2, ..., V_k. Select one vertex $v_i \in V_i$ from each component and form the set $U = \{v_1, v_2, \ldots, v_k\}$. Clearly, $|U| = k$, and no two vertices in U are connected, implying that the subgraph induced by U in G is disconnected. This contradicts the condition that every k-vertex subgraph is connected. Therefore, no k-VCP solution exists. □

The conditions in Propositions 1 and 2 are inherent feasibility conditions of the proposed master–slave partitioning problem. They are derived directly from the structural requirements specified in Definition 1. If either condition is violated, no partition satisfying the required master–slave structure can be constructed under the current problem formulation.

These results are illustrated by two examples in Figure 2. As shown in (a), vertices 5, 7, and 11 each form a star-shaped subgraph with all their neighbors. The subgraph induced by {5,7,11} forms the master block (red dashed box), while the subgraphs induced by {1,2,3,6}, {4,9,10,14,15}, and {8,12,13,16,17} form the slave blocks (blue dashed boxes), resulting in a master–slave graph. As shown in (b), every subgraph induced by any three vertices is connected. Consequently, no 3-VCP solution exists, and the graph cannot be partitioned into 4 blocks forming a master–slave graph with one master block and three slave blocks.



(**a**) With a 3-VCP solution (**b**) without a 3-VCP solution

Figure 2. Examples illustrating the master–slave graph partitioning.

4.2. The Main Idea of Partitioning a Graph into Master–Slave Blocks

Building upon the theoretical foundation and necessary conditions discussed in Section 4.1, we propose a master–slave graph partitioning approach. The proposed method is designed for graphs that satisfy the necessary conditions. For dense or highly interconnected graphs with pervasively distributed dependencies, a valid master–slave partition may not exist, rendering the proposed method inapplicable to such scenarios. This method draws inspiration from the solution strategies of the k-VCP. Traditional k-VCP approaches typically identify a minimal vertex cut set using maximum flow techniques, such that removing these vertices splits the graph into two disconnected subgraphs. These subgraphs can then be recursively partitioned until k subgraphs are obtained. Alternatively, some methods attempt to directly divide the graph into at least k disconnected subgraphs. For instance, Zhou et al. [34] select high-degree vertices as an initial candidate set and iteratively refine it by adding or removing vertices to minimize its size while maintaining at least k independent subgraphs. This idea provides valuable insights for designing the initial stage of our partitioning method.

The proposed partitioning method consists of two stages, initial partition generation and fine-tuning, as illustrated in Figure 3. In the initial stage, a heuristic approach is used to partition a graph into blocks with a master–slave structure. To guide this process, each vertex is scored based on its potential to be included in the master block. A set of vertices with scores exceeding a predefined threshold is chosen to constitute the master block. The remaining graph is then divided into $k-1$ subgraphs, which serve as the slave blocks. During the fine-tuning stage, the initial blocks are iteratively refined through master reduction and slave rebalancing strategies. Master reduction selectively relocates vertices to minimize the load of the master block, while slave rebalancing redistributes vertices among the slave blocks to achieve balanced loads. Both strategies are guided by the proposed partition evaluation function.

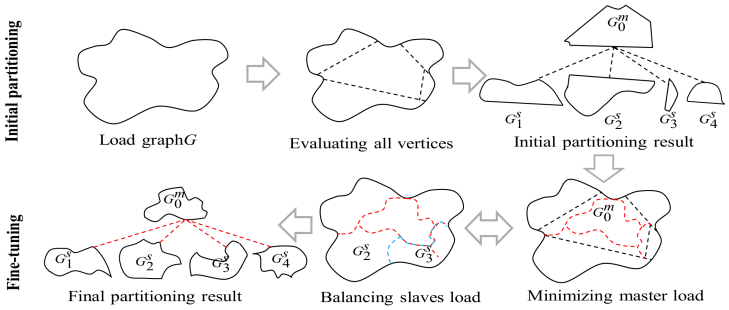


Figure 3. The main idea of building the target master–slave graph.

4.3. Initial Partition Block Generation Scheme

For a given load graph, the initial partitioning aims to generate a master block and $k-1$ slave blocks. A key challenge is selecting vertices with strong graph-separating capability and low computational overhead as candidates for the master block. To address this, we consider both structural and computational properties. Structurally, high-degree vertices often act as hubs connecting multiple subgraphs, and removing them tends to split the graph into more disconnected components. Computationally, vertices with low computational and communication costs help reduce bottlenecks when assigned to the master block. Additionally, certain low-degree vertices that bridge dense regions may also be good candidates for the master block.

Based on the above considerations, we design a scoring function to evaluate the priority of each vertex for inclusion in the master block. Formally, for each vertex $v \in V_L$ of a load graph $G_L = (V, E, w_L, \omega_L)$, the scoring function is defined as follows:

$$score(v) = \beta \frac{d(v)}{d_{max}} + \gamma\left(1 - \frac{C_A(v)}{C_{max}}\right) + \eta\left(1 - \frac{2|\{(u,v)|u \in N(v)\}|}{d(v)(d(v)-1)}\right) \tag{4}$$

where $\beta, \gamma, \eta > 0$ with $\beta + \gamma + \eta = 1$ are weighting factors, $d(v)$ is the degree of vertex v, d_{max} is the maximum vertex degree in graph G_L, $N(v)$ is the set of neighbors of vertex v, $C_A(v)$ is the computational cost of vertex v, and C_{max} is the maximum computational cost of a vertex.

The scoring function $score(v)$ combines a vertex's local structural features and computational cost to determine its priority as a candidate vertex for the master block. Vertices with higher scores tend to split the graph effectively while maintaining low computational overhead. Thus, vertices with scores above a threshold s_{min} are chosen to form a subgraph S. Rather than using a fixed absolute value, s_{min} is set adaptively as the top 15% quantile of the vertex score distribution. For a fixed threshold, adjusting the weights β, γ, η enables different partitioning preferences; for example, setting $\beta = 1$ makes the selection rely solely on vertex degree, as in [34]. The number of disconnected subgraphs in the remaining subgraph $G_L \backslash S$ is then examined. If the remaining graph contains more than $k-1$ connected subgraphs, the two smallest subgraphs are merged; if fewer, the highest-scoring vertex in the maximum-load subgraph is moved to the master block to further split the subgraph, until exactly $k-1$ disconnected subgraphs are obtained. The subgraph S forms the master block, and the remaining $k-1$ subgraphs serve as slave blocks, yielding the initial master–slave partitioning result. The corresponding pseudocode is presented in Algorithm 1. In terms of computational complexity, IP-MST requires $O\left(|V| + |E| + \sum_{v \in V} d(v)^2 + k^2\right)$ time under an adjacency-list representation, where $d(v)$ is the degree of vertex v, and k is the number of target blocks. The overall space complexity of IP-MST is $O(|V| + |E|)$.

Algorithm 1: Initial partitioning for master–slave topology, IP-MST.

Input: $G_L = (V, E, w_L, \omega_L)$, k, s_{min}; //G_L denotes the load graph, k is the target number of blocks, and s_{min} is the predefined threshold

Output: $MSP_{init} = \{G_0^m, G_1^s, G_2^s, \ldots, G_{k-1}^s\}$; //$MSP_{init}$ denotes the initial partition consisting of one master block and $k-1$ slave blocks

$V_0^m \leftarrow \emptyset$; //initialization
for $v \in V$
  $s \leftarrow score(v)$; //according to Equation (4)
  if $s > s_{min}$
    $V_0^m \leftarrow$add_vertex(v);
  end if
end for
$G_0^m \leftarrow$construct_master_block(G_L, V_0^m);
$\pi \leftarrow$set_of_subgraphs$(G_L \backslash G_0^m)$
while $|\pi| < k-1$ //increase the number of subgraphs
  $G_1 \leftarrow$select_max_load_subgraph(π);
  if $|V(G_1)| < 2$ //G_1 contains fewer than two vertices
    return failure
  end if
  $S_{max} \leftarrow 0$;
  for $v \in V_1$
    if $s_{max} < score(v)$
      $s_{max} \leftarrow score(v)$;
      $v' \leftarrow v$;
    end if
  end for
  $G_0^m \leftarrow$add_vertex(G_L, G_0^m, G_1, v');
  if is_not_connected(G_1)
    $\pi \leftarrow$update_set_of_subgraphs (π, G_1);
  end if
end while
while $|\pi| > k-1$ //decrease the number of subgraphs
  $G_1, G_2 \leftarrow$select_smallest_two_subgraphs (π);
  $G'_1 \leftarrow$merge_subgraphs (G_1, G_2);
  $\pi \leftarrow$update_set_of_subgraphs (π, G_1, G_2, G'_1);
end while
$\{G_1^s, G_2^s, \ldots, G_{k-1}^s\} \leftarrow$generate_slave_blocks$(\pi)$;
$MSP_{init} \leftarrow \{G_0^m, G_1^s, G_2^s, \ldots, G_{k-1}^s\}$;
return MSP_{init};

4.4. Target Partition Block Quality Optimization Strategies

The initial master–slave partition often contains an oversized master block and suffers from imbalanced loads among the slave blocks, necessitating further optimization. To guide this process, a partition evaluation function is introduced to quantify how well the current partition meets the target objectives. Based on this function, distinct optimization strategies are applied to the master block and the slave blocks.

4.4.1. Graph Partitioning Evaluation Function

According to the dual objective of a minimal master block and balanced slave blocks, we define the partition quality function $Q(MSP(G_L,k))$. Specifically, given a load graph $G_L = (V_L, E_L, w_L, \omega_L)$, which is partitioned into k blocks, denoted as $MSP(G_L,k) = \{G_0^m, G_1^s, G_2^s, \ldots, G_{k-1}^s\}$, where $G_0^m = (V_0^m, E_0^m)$ represents the master block,

and each $G_i^s = (V_i^s, E_i^s)$ for $i = 1, 2, \ldots, k-1$ corresponds to a slave block, the partition quality is measured by the function $Q(MSP(G_L, k))$:

$$Q(MSP(G_L, k)) = \tau \frac{\sigma_s}{\overline{C}_A} + (1 - \tau) \frac{C_A(G_0^m)}{C_{total}} \tag{5}$$

where τ is an adjustable weight that balances the two objectives: minimizing the load of the master block and maintaining load balance among the slave blocks, and $0 < \tau < 1$. $\sigma_s = \left(\sum_{i=1}^{k-1} \left(C_A(G_i^s) - \overline{C}_A \right)^2 / (k-1) \right)^{1/2}$ denotes the standard deviation of slave loads, $\overline{C}_A = \left(C_{total} - C_A(G_0^m) \right) / (k-1)$ is the average load of all slave blocks. $C_A(G_0^m)$ is the load of the master block, $C_A(G_i^s)$ is the load of the slave block G_i^s, and $C_{total} = C_A(G_0^m) + \sum_{i=1}^{k-1} C_A(G_i^s)$ denotes the total load under the partition $MSP(G_L, k)$. According to Equation (5), the metric Q is a non-negative cost-based measure for evaluating graph partitioning quality, where a lower value indicates a better partitioning result. For a fixed total cost C_{total}, reducing either the master load $C_A(G_0^m)$ or the standard deviation of the slave load σ_s decreases Q, with a smaller Q indicating better partitioning quality. Based on this metric, the partition optimization is designed to achieve two objectives: minimizing the master load and reducing the load variance among slave blocks. These objectives are implemented through Master Reduce Operation (MRO) and Slave Rebalance Operation (SRO), respectively. For a given partition $MSP(G_L, k)$, applying an MRO or SRO yields a refined partition $MSP'(G_L, k)$, and an operation is accepted if $Q(MSP'(G_L, k)) < Q(MSP(G_L, k))$. MRO and SRO are applied alternately to iteratively improve the partition quality until a termination condition is met. The pseudocode of this procedure is given in Algorithm 2. If the refinement process performs I iterations, the time complexity of PR-MMBS is $O(I(T_{MRO} + T_{SRO}))$, where T_{MRO} and T_{SRO} denote the time complexities of MRO and SRO, respectively. The overall space complexity of PR-MMBS is $O(|V| + |E|)$.

Algorithm 2: Partition refinement for minimizing master and balancing slaves, PR-MMBS.

Input: $G_L = (V, E, w_L, \omega_L)$, MSP_{init}; //G_L denote the load graph, $MSP_{init} = \{G_0^m, G_1^s, G_2^s, \ldots, G_{k-1}^s\}$ is the initial partition
Output: $MSP_{opt} = \{G'^m_0, G'^s_1, G'^s_2, \ldots, G'^s_{k-1}\}$;//$MSP_{opt}$ denotes the target partition

```
Q_best←Q(MSP_init), sign←True, MSP_opt←MSP_init;
while sign
    sign←False;
    MSP'←MRO(G_L, MSP_opt); //master reduce
    if Q(MSP') < Q_best
        sign←True;
        Q_best←Q(MSP');
        MSP_opt←MSP';
    end if
    MSP'←SRO(G_L, MSP_opt); //slaves rebalance
    if Q(MSP') < Q_best
        sign←True;
        Q_best←Q(MSP');
        MSP_opt←MSP';
    end if
end while
return MSP_opt;
```

Proposition 3. *Repeatedly applying Master Reduce Operation (MRO) and Slave Rebalance Operation (SRO) alternately from an initial partition decreases the partition quality metric Q at each step and converges in a finite number of operations.*

Proof of Proposition 3. Let $MSP(G_L, k)$ denote a master–slave partition of a load graph G_L, and let P be the set of all partitions. Since the number of vertices in G_L is finite, the size of P is also finite. Each application of MRO or SRO strictly decreases the partition quality metric Q. Therefore, Q forms a strictly decreasing sequence with a finite number of elements and must terminate after a finite number of operations. □

4.4.2. Master Load Minimization Strategy

To minimize the master block, we propose the Master Reduce Operation (MRO). This strategy reduces the master load by migrating vertices from the master block to slave blocks while preserving the topology constraints of the master–slave graph. A master vertex that is connected to multiple slave blocks cannot be moved, as this could disrupt the master–slave structure of the original partition. Therefore, only vertices connected to at most one slave block are considered movable. Let a partition $MSP = \{G_0^m, G_1^s, G_2^s, \ldots, G_{k-1}^s\}$, For a vertex $v \in V_0^m$, if there exists an $i \in \{1, 2, \ldots, k-1\}$ such that $N(v) \subseteq V_0^m \cup V_i^s$, then v is a candidate vertex for migration. A migration operation $o_{MR}(v)$ is defined as moving a candidate vertex from G_0^m to a slave block G_i^s, resulting in a new partition $MSP' = \{G'^m_0, G_1^s, G_2^s, \ldots, G'^s_i, \ldots, G_{k-1}^s\}$, denoted as $o_{MR}(v) : MSP \to MSP'$, where G'^m_0 is obtained by removing v and related edges from G_0^m, and G'^s_i is obtained by adding v and related edges to G_i^s. For a vertex migration operation $o_{MR}(v) : MSP \to MSP'$, if $\Delta Q(o_{MR}(v)) = Q(MSP) - Q(MSP') > 0$, then the migration of vertex v is considered valid. For all candidate vertices, the vertex corresponding to $o^* = \arg\max_{o \in \{o_{MR}\}} \Delta Q(o)$ is selected in each iteration. If $\Delta Q(o^*) > 0$, we perform the migration of the vertex and update the partition. Otherwise, the procedure stops, indicating that no further beneficial migration is available in the master block and the current partition has reached a local optimum. The pseudocode of this process is presented in Algorithm 3. With incremental updates of the affected block loads and the partition quality metric, the time complexity of MRO is $O\left(|V|^2 + |E|\right)$ in the worst case. The overall space complexity of MRO is $O(|V| + |E|)$.

Algorithm 3: Master Reduce Operation, *MRO*.

Input: $G_L = (V, E, w_L, \omega_L)$, MSP_{init};//G_L denotes the load graph, $MSP_{init} = \{G_0^m, G_1^s, G_2^s, \ldots, G_{k-1}^s\}$ is the initial partition
Output: $MSP_{opt} = \{G'^m_0, G'^s_1, G'^s_2, \ldots, G'^s_{k-1}\}$;//$MSP_{opt}$ denotes the improved partition

```
MSP_opt ← MSP_init;
V_cand ← select_candidate_vertices(G_L, MSP_opt);
while |V_cand|
    v ← arg max_{v∈V_cand} ΔQ(o_MR(v));
    G_i^s ← connected_slave(v, MSP_opt);
    V_cand ← V_cand \ {v};
    if ΔQ(o_MR(v)) > 0
        MSP_opt ← update(MSP_opt, G_i^s, v);
    end if
end while
return MSP_opt;
```

Proposition 4. *Let a partition be* $MSP(G_L,k)=\{G_0^m,G_1^s,G_2^s,\dots,G_{k-1}^s\}$. *Under the constraints of Equations (2) and (3), the computational cost of the master block* $G_0'^m$ *obtained by applying the MRO strategy satisfies:* $\max\{\min_{v\in V}C_A(v),\sum_{v\in V}C_A(v)-(k-1)(1+\varepsilon)\overline{C}_A\}\le C_A(G_0'^m)\le\sum_{v\in V}C_A(v)-(k-1)\min_i C_A(G_i^s)$.

Proof of Proposition 4. Given a load graph $G_L=(V,E,w_L,\omega_L)$ and a partition $MSP(G_L,k)=\{G_0^m,G_1^s,G_2^s,\dots,G_{k-1}^s\}$, applying the MRO strategy produces a new partition $MSP'=\{G_0'^m,G_1^s,G_2^s,\dots,G_i'^s,\dots,G_{k-1}^s\}$. The master block in MSP' contains at least one vertex; hence, $C_A(G_0'^m)\ge\min_{v\in V}C_A(v)$. Let the total computational cost of all vertices in the graph be $C_{total}=\sum_{v\in V}C_A(v)$. Under the load-balance constraint of Equation (2), each slave block has cost at most $(1+\varepsilon)\overline{C}_A$, so that $\sum_{i=1}^{k-1}C_A(G_i^s)\le(k-1)(1+\varepsilon)\overline{C}_A$. It follows that $C_A(G_0'^m)\ge\max\{\min_{v\in V}C_A(v),\sum_{v\in V}C_A(v)-(k-1)(1+\varepsilon)\overline{C}_A\}$. If the master minimization constraint is not enforced, let $\min_i C_A(G_i^s)$ denote the minimum load among $k-1$ slave blocks. In the extreme case of maximizing the master block, all vertices except those belonging to the $k-1$ smallest slave blocks can be assigned to the master, yielding $(G_0'^m)\le\sum_{v\in V}C_A(v)-(k-1)\min_i C_A(G_i^s)$. □

4.4.3. Slave Load Balancing Strategy

To further improve the load balance among slave blocks, we propose the Slave Rebalance Operation (SRO) strategy. The strategy selects a set of vertices as a cut to split the heaviest-loaded slave block into two sub-blocks, after which the smaller one is reassigned to the lightest-loaded slave block. This process iteratively improves load distribution across slave blocks.

Specifically, consider a load graph $G_L=(V,E,w_L,\omega_L)$ and a partition $MSP(G_L,k)=\{G_0^m,G_1^s,G_2^s,\dots,G_{k-1}^s\}$. In conventional approaches, directly migrating vertices between slave blocks often introduces new cut edges between slaves and may violate topology constraints of the master–slave graph. To address this issue, we propose a new migration strategy. Without loss of generality, let the slave block $G_i^s=(V_i^s,E_i^s)$ be the one with the minimum load and the slave block $G_j^s=\left(V_j^s,E_j^s\right)$ be the one with the maximum load, where $1\le i\le j\le k-1$. The slave block G_j^s is first processed using the Goldberg–Rao algorithm [37] to identify a minimal set of cut vertices $V_c\subset V_j^s$. Removing V_c partitions G_j^s into two disconnected subgraphs, $G_{j_1}^s$ and $G_{j_2}^s$, with $G_{j_1}^s$ being the smaller one. The smaller subgraph $G_{j_1}^s$ is then reassigned to the slave block G_i^s, yielding an updated slave block $G_i'^s$, while V_c and its associated edges are migrated to the master block G_0^m, resulting in $G_0'^m$. The new partition is thus given by $SP'=\left\{G_0'^m,G_1^s,\dots,G_i'^s,\dots,G_{j-1}^s,G_{j_2}^s,G_{j+1}^s,\dots,G_{k-1}^s\right\}$, which still forms a master–slave structure. This operation is denoted as $o_{SR}\left(G_{j_1}^s\right):MSP\to MSP'$. To ensure that this migration improves the overall partition quality, we evaluate it using the metric Q. The migration is valid if $\Delta Q\left(o_{SR}\left(G_{j_1}^s\right)\right)=Q(MSP)-Q(MSP')$. The pseudocode of this procedure is provided in Algorithm 4. The time complexity of SRO is $O(k+n_S+m_S+T_{GR}(n_S,m_S))$, where n_S and m_S denote the numbers of vertices and edges in the processed slave block, and T_{GR} denotes the time complexity of the Goldberg–Rao procedure. The overall space complexity of SRO is $O(|V|+|E|)$.

Proposition 5. *Consider a master–slave partitioning* $MSP(G_L,k)=\{G_0^m,G_1^s,G_2^s,\dots,G_{k-1}^s\}$ *of graph* G_L; *let* $\overline{C}_A$ *be the average load of the k blocks. Suppose that a slave block* G_j^s *contains a subgraph* $H\subseteq G_j^s$ *such that* $C_A(H)=\overline{C}_A+\delta$ *for some* $\delta>0$, *and H cannot be migrated out during the optimization process. Then, the standard deviation of the slave block loads satisfies* $\sigma_s\ge\delta/(k-2)^{1/2}$.

Proof of Proposition 5. Consider the partition $MSP(G_L,k)=\{G_0^m,G_1^s,G_2^s,\dots,G_{k-1}^s\}$; let the loads of the slave blocks be $l_1,l_2,\dots,l_{k-1}$, and the average load is $\overline{C}_A$. Without loss of

generality, assume that $l_1 = C_A\left(G_j^s\right) \geq C_A(H)$. Since the total load of all slave blocks satisfies $\sum_{i=1}^{k-1} l_i = (k-1)\overline{C}_A$, it follows that the sum of the remaining $k-2$ slave blocks is $\sum_{i=2}^{k-1} l_i = (k-1)\overline{C}_A - l_1 = (k-2)\overline{C}_A - \delta$. To minimize the variance, these $k-2$ slave blocks should be equal, so let $l_2 = l_3 = \cdots = l_{k-1} = \overline{C}_A - \delta/(k-2)$. Then, the variance of the slave block loads is $\sigma^2 = \sum_{i=1}^{k-1}\left(l_i - \overline{C}_A\right)^2/(k-1) = \delta/(k-2)^{1/2}$. Hence, the standard deviation of the slave block loads satisfies $\sigma_s \geq \delta/(k-2)^{1/2}$. □

Algorithm 4: Slave Rebalance Operation, *SRO*.

Input: $G_L = (V, E, w_L, \omega_L)$, MSP_{init};//G_L denotes the load graph,
$MSP_{init} = \{G_0^m, G_1^s, G_2^s, \ldots, G_{k-1}^s\}$ is the initial partition
Output: $MSP_{opt} = \{G'^m_0, G'^s_1, G'^s_2, \ldots, G'^s_{k-1}\}$;//$MSP_{opt}$ denotes the improved partition

$MSP_{opt} \leftarrow MSP_{init}$, $Q_{best} \leftarrow Q(MSP_{init})$;
S_{min}, $S_{max} \leftarrow$ select_slaves(MSP_{opt});
if is_connected(S_{max})
  G_{min}, G_{max}, $V_c \leftarrow$ Goldberg_Rao(S_{max});
  $S_{min} \leftarrow S_{min} \cup G_{min}$, $S_{max} \leftarrow S_{max} \backslash G_{min}$;
  $G'^m_0 \leftarrow$ update_master(G_0^m,V_c);
  $MSP' \leftarrow$ update(MSP_{opt}, S_{min}, S_{max}, G'^m_0);
  if $Q(MSP') < Q_{best}$
    $MSP_{opt} \leftarrow MSP'$;
  end if
else
  $G_{min} \leftarrow$ select_min_component(S_{max});
  $S_{min} \leftarrow S_{min} \cup G_{min}$, $S_{max} \leftarrow S_{max} \backslash G_{min}$;
  $MSP' \leftarrow$ update(MSP_{opt}, S_{min}, S_{max});
  if $Q(MSP') < Q_{best}$
    $MSP_{opt} \leftarrow MSP'$;
  else
    G'_{min}, G'_{max}, $V'_c \leftarrow$ Goldberg_Rao(G_{min});
    $S_{min} \leftarrow S_{min} \cup G'_{min}$, $S_{max} \leftarrow S_{max} \backslash G'_{min}$;
    $G'^m_0 \leftarrow$ update_master(G_0^m,V'_c);
    $MSP' \leftarrow$ update(MSP_{opt}, S_{min}, S_{max}, G'^m_0);
    if $Q(MSP') < Q_{best}$
      $MSP_{opt} \leftarrow MSP'$;
    end if
  end if
end if
return MSP_{opt};

5. Experimental Evaluation

To evaluate the effectiveness of the proposed master–slave graph partitioning method (MSGP), we conducted four types of experiments from different perspectives: (1) performance evaluation across diverse graph datasets; (2) comparative analysis against representative traditional methods, focusing on partition quality, master block load control, and slave load balance; (3) ablation studies assessing the contributions of individual modules and parameter settings; (4) stability analysis evaluating the robustness of our proposed partitioning method. All experiments were performed on graph datasets that satisfied the proposed necessary conditions.

5.1. Experimental Setup

All algorithms were implemented in Python 3.12 using NetworkX 3.5. The experiments were conducted using four servers configured in a master–slave architecture. Each server

ran Ubuntu 24.04 and was equipped with a 64-bit Intel Xeon CPU (8 cores @ 1.9 GHz), 128 GB of memory, and a 2 TB SSD. In our experiments, multiple logical blocks are co-located on each physical machine, and inter-block communication within the same machine is mediated through local shared-memory buffers rather than the network. The execution time measurements therefore primarily reflect computation cost and local synchronization overhead rather than network communication effects. Network latency, bandwidth contention, and cross-machine communication delays are not explicitly modeled in the current experimental environment.

The graph datasets used in the experiments were downloaded from SNAP and the Network Repository, including fb-pages-government, ca-HepTh, fb-pages-company, soc-gemsec-RO, loc-Gowalla, and com-DBLP, as summarized in Table 2. These datasets cover a wide range of scales and topological types. PageRank, Louvain, BFS, Kruskal, GAT, and SSSP were selected as downstream application algorithms for constructing the load graphs. The evaluation metrics include the normalized standard deviation of slave loads, the cut-edge ratio, the load ratio of the master block, and the proposed partition quality metric Q (a non-normalized composite metric that jointly quantifies master block overhead and slave load balance).

Table 2. Experimental dataset.

Name	|V|	|E|	Diameter	Clustering Coefficient
fb-pages-government	7057	89,455	10	0.4109
ca-HepTh	9877	25,998	17	0.4714
fb-pages-company	14,115	52,311	15	0.2392
soc-gemsec-RO	41,773	125,826	13	0.0912
loc-Gowalla	196,591	950,327	14	0.2367
com-DBLP	317,080	1,049,866	21	0.6324

The proposed master–slave graph partitioning method, MSGP, integrates IP-MST and PR-MMBS. To evaluate its performance, we conducted comparative experiments with four representative algorithms: Metis [4], FLS [34], LocalFlow [37], and CNC [38]. These baselines were selected to represent the major categories of existing graph partitioning approaches: Metis is a widely used multilevel partitioner serving as the de facto standard baseline; FLS represents fast local-search-based methods; LocalFlow represents flow-based approaches; and CNC represents connectivity-optimized partitioning.

For reproducibility, the main parameter settings used in the experiments are specified here. For MSGP, $\tau = 0.6$, $\beta = 0.4$, $\gamma = 0.3$, $\eta = 0.3$, while $s_{min} = 15\% \times |V|$. These settings were fixed across the experiments unless otherwise specified. For each comparative experiment, the same target number of blocks k was used for MSGP and all baseline methods. Metis was run with its default parameter configuration. For LocalFlow, the seed vertex was selected as the highest-degree vertex in the corresponding subgraph. For FLS, the "remove" and "add" actions were randomly selected with probabilities of 0.8 and 0.2, respectively. For CNC, the random seed was fixed at 42. No dataset-specific parameter tuning was performed for the baseline methods.

5.2. Experiments Analysis

Experiment 1. *Comprehensive Evaluation of the Proposed MSGP.*

To comprehensively evaluate the proposed graph partitioning method MSGP, we design experiments from multiple perspectives. The evaluation metrics include: (1) balance of slave partitions, measured by the normalized standard deviation (NSD) and the maximum deviation ratio (DEV); (2) cut-edge ratio (R_c), which reflects the communication cost between the master block and the slave blocks; (3) load proportion of the master block (R_m); and (4) proposed partition quality metric Q. For each dataset, MSGP is executed with varying numbers of blocks, k = 8, 16, 24, ..., 64. Since the master–slave structure requires one master block and at least three slave blocks to demonstrate meaningful parallel execution, k = 8 is chosen as the minimum partition count. Smaller values of k (e.g., k = 2 or 3) would leave too few slave blocks to observe the benefits of parallel processing. All corresponding metrics are recorded. As illustrated in Figure 4 and Table 3, the partition quality is jointly influenced by both the underlying graph topology and the characteristics of the application algorithms. In particular, iterative algorithms such as PageRank and Louvain exhibit more stable and higher-quality partitions under MSGP. This can be attributed to their relatively regular communication patterns and iterative computation structures, which are well aligned with the master–slave topology. Furthermore, as the number of blocks increases, the partition quality initially improves due to finer-grained load distribution, and then gradually stabilizes. As shown in Table 3, increasing k leads to a moderate rise in the master load ratio R_m and a slight increase in the number of cut edges between the master and slave blocks. Meanwhile, the balance among slave partitions exhibits only minor fluctuations, resulting in a slight degradation in the overall partition quality. These observations indicate that MSGP achieves a structurally adaptive trade-off between computation and communication, rather than optimizing either objective in isolation.

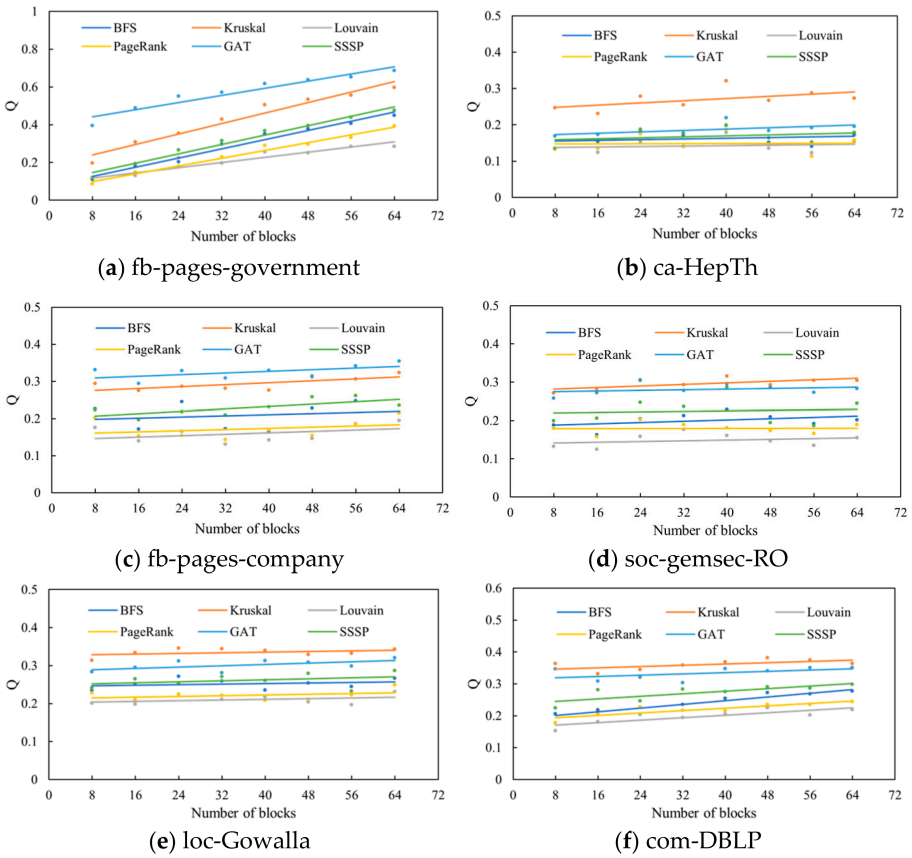
Experiment 2. *Comparison with Traditional Partitioning Methods.*

(a) Performance Comparison with Traditional Methods

The proposed MSGP method was compared against four graph partitioning methods: Metis, FLS, LocalFlow, and CNC. The comparison uses two categories of metrics. The first category consists of approach-independent metrics: load proportion of the master block (R_m), the communication cost ratio (R_c), normalized standard deviation of slave loads (NSD), and maximum deviation ratio (DEV), which can be computed for any partitioning result regardless of its structural assumptions. The second category is the master–slave-specific quality metric Q, which directly evaluates how well a partition conforms to the star-shaped topology proposed in this work. For each dataset, the partition quality Q was recorded for block counts k= 8, 16, 24, ..., 64. A detailed comparison of other metrics was then conducted on the fb-pages-government dataset at k= 32. As summarized in Figure 5 and Table 4, on the approach-independent metrics, MSGP achieves the lowest NSD and DEV across all datasets, indicating substantially better slave load balance than the baselines. On the master–slave-specific metric Q, MSGP consistently achieves the lowest values, reflecting the high quality of the constructed star-shaped structure. These results indicate its superior overall quality, achieved by effectively balancing the master load against slave load uniformity. Furthermore, the DEV results reveal that while FLS, LocalFlow, and CNC can produce a master–slave graph structure, they lead to severe imbalance among the slave blocks.

Table 3. R_m, R_c, *NSD* and *DEV* values under PageRank.

		8	16	24	32	40	48	56	64
fb-pages-government	R_m	0.153	0.159	0.166	0.165	0.167	0.168	0.174	0.172
	R_c	0.274	0.275	0.276	0.276	0.277	0.277	0.280	0.279
	NSD	0.001	0.011	0.112	0.171	0.248	0.293	0.328	0.361
	DEV	0.001	0.016	0.218	0.584	0.332	0.500	0.642	0.589
ca-HepTh	R_m	0.152	0.158	0.160	0.161	0.162	0.163	0.163	0.163
	R_c	0.183	0.185	0.186	0.187	0.187	0.186	0.187	0.189
	NSD	0.079	0.115	0.001	0.133	0.153	0.013	0.048	0.134
	DEV	0.089	0.153	0.003	0.182	0.301	0.020	0.068	0.206
fb-pages-company	R_m	0.141	0.148	0.151	0.151	0.153	0.153	0.153	0.153
	R_c	0.163	0.163	0.164	0.165	0.164	0.169	0.168	0.169
	NSD	0.202	0.064	0.174	0.128	0.189	0.222	0.264	0.187
	DEV	0.491	0.082	0.300	0.200	0.417	0.477	0.705	0.354
soc-gemsec-RO	R_m	0.172	0.175	0.178	0.179	0.179	0.180	0.180	0.180
	R_c	0.194	0.194	0.195	0.195	0.197	0.196	0.198	0.199
	NSD	0.293	0.302	0.069	0.226	0.208	0.041	0.091	0.184
	DEV	0.479	0.311	0.089	0.380	0.513	0.059	0.139	0.293
loc-Gowalla	R_m	0.127	0.128	0.130	0.131	0.131	0.131	0.132	0.132
	R_c	0.147	0.146	0.148	0.147	0.147	0.148	0.149	0.149
	NSD	0.182	0.101	0.204	0.028	0.131	0.213	0.118	0.059
	DEV	0.326	0.143	0.417	0.042	0.191	0.396	0.209	0.094
com-DBLP	R_m	0.114	0.123	0.127	0.128	0.127	0.129	0.128	0.128
	R_c	0.195	0.197	0.198	0.198	0.198	0.199	0.199	0.199
	NSD	0.157	0.155	0.118	0.298	0.255	0.130	0.205	0.277
	DEV	0.255	0.245	0.168	0.635	0.686	0.213	0.400	0.590



(**a**) fb-pages-government (**b**) ca-HepTh

(**c**) fb-pages-company (**d**) soc-gemsec-RO

(**e**) loc-Gowalla (**f**) com-DBLP

Figure 4. Partition quality Q under different application algorithms.

(b) Comparison of PageRank Execution Times across Partitioning Methods

To evaluate the impact of different partitioning strategies on computational efficiency, we measured the execution time of the PageRank algorithm on the com-DBLP dataset under varying block counts k= 8, 16, 24, …, 64. Note that all experiments were conducted on the same platform described in Section 5.1, where multi-node execution was simulated based on this limited physical infrastructure. The proposed MSGP method was compared with four representative approaches (Metis, FLS, LocalFlow, and CNC), as well as a non-partitioned baseline (M0). For each partitioning result, PageRank was executed five times, and the average execution time was recorded, as shown in Figure 6. The standard deviation across repetitions was below 5% of the mean in all cases, indicating stable and statistically robust timing measurements. The partitioning cost is a one-time preprocessing overhead. As illustrated in Figure 6, all partitioning methods achieve significant speedups compared to the non-partitioned baseline. MSGP consistently achieves the lowest execution time under our co-located experimental setup. We note that these measurements reflect computation and local synchronization costs. The ranking of methods by execution time in Figure 6 is consistent with their ranking by the partition quality metric Q in Table 3. Across all block counts k, MSGP, which achieves the lowest Q values, consistently exhibits the shortest execution time, while methods with higher Q (FLS, LocalFlow, CNC) correspondingly require longer execution times. These results corroborate the empirical correlation between Q and PageRank execution performance within our specific co-located experimental setup.

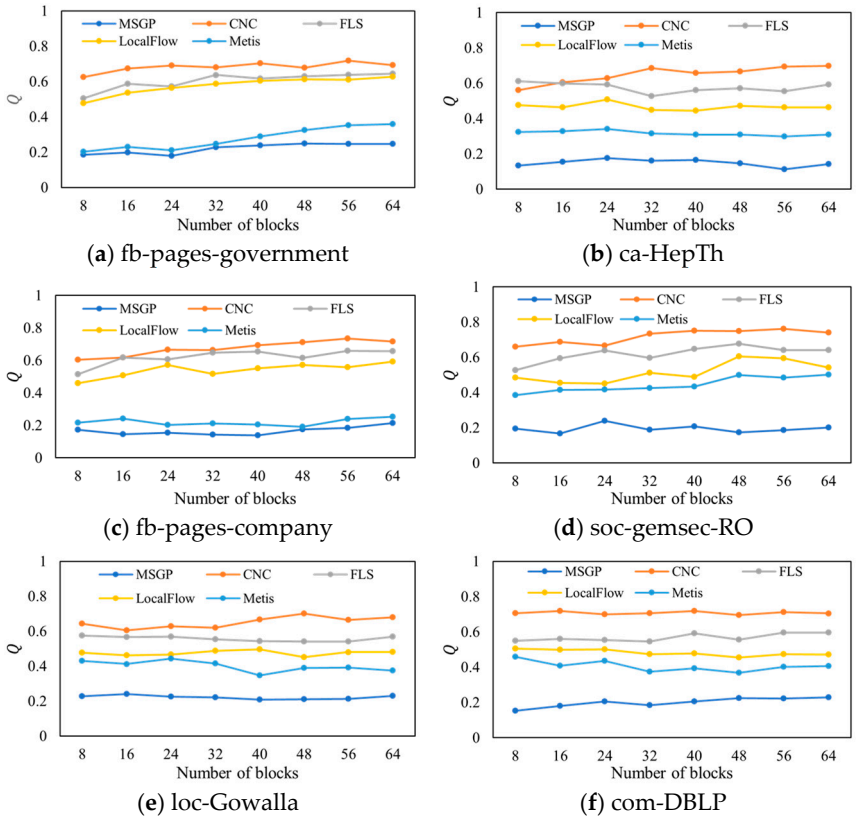


Figure 5. Partition quality of the compared algorithms across different datasets.

Table 4. R_m, R_c, *NSD* and *DEV* values of the compared algorithms.

		R_m	R_c	*NSD*	*DEV*
fb-pages-government	MSGP	0.165	0.276	0.171	0.584
	Metis	-	0.088	0.612	2.724
	CNC	0.130	0.219	0.849	31.848
	FLS	0.235	0.263	0.745	29.859
	LocalFlow	0.295	0.314	0.543	21.483
ca-HepTh	MSGP	0.161	0.187	0.133	0.182
	Metis	-	0.157	0.527	2.163
	CNC	0.173	0.192	0.948	175.197
	FLS	0.161	0.248	0.748	29.673
	LocalFlow	0.090	0.171	0.548	21.876
fb-pages-company	MSGP	0.151	0.159	0.128	0.200
	Metis	-	0.131	0.486	1.270
	CNC	0.127	0.190	0.830	29.929
	FLS	0.109	0.155	0.811	25.919
	LocalFlow	0.122	0.171	0.794	25.356
soc-gemsec-RO	MSGP	0.179	0.199	0.226	0.680
	Metis	-	0.145	0.339	0.512
	CNC	0.157	0.187	0.897	35.975
	FLS	0.081	0.152	0.845	30.967
	LocalFlow	0.150	0.191	0.762	29.942
loc-Gowalla	MSGP	0.131	0.147	0.028	0.042
	Metis	-	0.143	0.723	7.738
	CNC	0.135	0.168	0.708	28.699
	FLS	0.093	0.141	0.587	25.638
	LocalFlow	0.010	0.024	0.570	24.679
com-DBLP	MSGP	0.128	0.198	0.298	0.635
	Metis	-	0.102	0.786	14.733
	CNC	0.072	0.103	0.869	33.927
	FLS	0.086	0.113	0.837	30.153
	LocalFlow	0.061	0.098	0.765	29.973

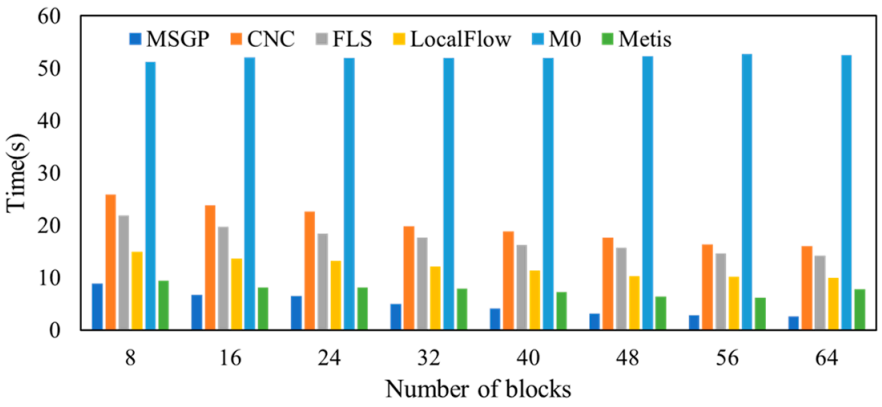


Figure 6. Comparison of algorithm execution time.

(c) Partitioning Time Analysis

To evaluate the preprocessing cost of MSGP, we measured the running time of IP-MST and PR-MMBS separately on six datasets with k = 8, 16, 24, ..., 64. The resulting partitions were then used for PageRank execution, and the corresponding execution times were measured separately. The recorded execution times are presented in Table 5. The

experimental results show that IP-MST incurs relatively stable costs across different values of k, while PR-MMBS accounts for most of the partitioning time. The refinement cost generally increases with k, although fluctuations occur for some datasets. In contrast, PageRank execution time decreases consistently as k increases across all six datasets. These results reveal a trade-off between partitioning cost and subsequent execution performance. Since graph partitioning is a preprocessing step, its cost can be amortized when the resulting partition is reused for multiple executions. For the com-DBLP dataset with $k = 32$, the full MSGP preprocessing pipeline takes approximately 11,076 s, while each PageRank execution saves about 50 s compared with the non-partitioned baseline. Thus, the preprocessing cost can be amortized after approximately 220 PageRank executions.

Table 5. Running time of partitioning and PageRank execution under different k values.

		8	**16**	**24**	**32**	**40**	**48**	**56**	**64**
fb-pages-government	IP-MST (s)	3.47	3.82	3.69	3.15	3.08	2.54	2.36	2.91
	PR-MMBS (s)	133.38	289.17	306.13	317.29	327.34	338.38	347.24	356.51
	PageRank (s)	4.04	3.05	2.98	2.30	1.87	1.45	1.27	1.18
ca-HepTh	IP-MST (s)	2.13	1.94	1.88	2.21	1.94	2.07	1.89	1.92
	PR-MMBS (s)	122.47	132.19	126.83	157.05	171.62	185.38	198.71	213.94
	PageRank (s)	3.47	2.42	2.35	1.83	1.49	1.20	1.05	0.97
fb-pages-company	IP-MST (s)	1.94	2.18	2.05	2.15	1.83	1.99	1.97	1.95
	PR-MMBS (s)	237.61	258.29	323.07	255.84	144.53	257.18	290.76	444.92
	PageRank (s)	2.97	2.10	2.03	1.60	1.34	1.13	1.02	0.94
soc-gemsec-RO	IP-MST (s)	3.68	3.75	3.72	3.86	3.93	4.12	4.21	4.27
	PR-MMBS (s)	853.42	905.17	998.68	1040.09	1122.73	1228.35	1087.91	1394.56
	PageRank (s)	4.30	3.10	3.03	2.33	1.91	1.52	1.31	1.19
loc-Gowalla	IP-MST (s)	128.05	129.12	128.18	127.25	128.32	127.38	127.45	128.52
	PR-MMBS (s)	6608.37	6714.52	6689.14	6825.79	6902.46	6987.23	7051.88	7126.05
	PageRank (s)	6.48	4.83	4.71	3.66	3.00	2.38	2.10	1.92
com-DBLP	IP-MST (s)	1866.42	1857.29	1845.24	1861.31	1847.37	1863.42	1848.46	1858.53
	PR-MMBS (s)	8972.61	9080.29	9047.83	9215.07	9304.58	9412.36	9358.94	9527.18
	PageRank (s)	8.89	6.72	6.56	5.05	4.11	3.18	2.8	2.59

(d) Relationship between Q and Execution Performance

To further examine the relationship between the proposed Q metric and actual application execution performance, we conducted a controlled experiment using four different partitioning configurations on the fb-pages-government dataset. PageRank was executed with 32 partitions under the same experimental configuration described in Section 5.1. The four configurations correspond to the initial partition, optimization without slave-load rebalancing, optimization without master-load reduction, and the complete MSGP method, respectively. Since the dataset, application algorithm, and partition count were fixed, the resulting partitions provide a controlled setting for comparing Q with execution time. The corresponding Q values and execution times are reported in Table 6. The results show that lower Q values are consistently associated with shorter execution times across the four partitioning configurations, providing empirical evidence that the proposed Q metric is correlated with PageRank execution performance under the tested conditions.

Table 6. Q values and execution times under different partitioning configurations.

Partitioning Configuration	Q	Execution Time (s)
A1	0.521	7.52 ± 0.06
A2	0.246	5.24 ± 0.05
A3	0.480	7.12 ± 0.05
A4	0.227	5.05 ± 0.04

Experiment 3. *Ablation studies.*

(a) Component Ablation of the Proposed Method.

To assess the contribution of each component in the proposed method, we designed four experimental variants: A1 (No Optimization), which generates the initial partition using only IP-MST; A2 (No Slave Rebalance), which disables the slave load balancing strategy during the optimization phase; A3 (No Master Reduce), which performs the optimization phase without reducing the master load; and A4 (Full Method), which executes the complete MSGP algorithm. All variants were evaluated on the fb-pages-government dataset using PageRank with $k = 32$. The results for metrics R_m, R_c, NSD, DEV, and Q are summarized in Table 7. The results show that A1 yields the poorest partition quality. A2 significantly improves the balance among slave blocks, while A3 reduces the master load but leaves slave blocks unbalanced. The full method, A4, produces a slightly larger master block than A3 and slightly less balanced slave blocks than A2, yet achieves the highest overall partition quality, demonstrating the complementary effect of the optimization components.

Table 7. Ablation study results of the proposed MSGP method.

	R_m	R_c	NSD	DEV	Q
A1	0.250	0.264	0.479	4.707	0.521
A2	0.256	0.268	0.101	0.285	0.246
A3	0.155	0.395	0.494	4.879	0.480
A4	0.165	0.379	0.171	0.584	0.227

(b) Parameter Sensitivity Analysis of τ

We further conducted a sensitivity analysis to evaluate the impact of the tuning factor τ on partitioning performance. This parameter balances the trade-off between master load minimization and slave load balancing in the quality metric Q. The analysis was performed on multiple graph datasets with τ set to 0.2, 0.4, 0.6, 0.8, and 1.0. For each setting, we recorded the NSD of slave block loads and the partition quality Q, as summarized in Table 8. The results indicate that $\tau = 0.6$ achieves the best partition quality. Although NSD is not optimal at this setting, it confirms that $\tau = 0.6$ effectively balances both objectives. In contrast, extreme values of τ overemphasize one objective, degrading overall partition quality.

(c) Parameter Sensitivity Analysis of β, γ, and η

We also conducted a sensitivity analysis to evaluate the impact of the weighting factors β, γ, and η on partitioning performance. Six combinations of these parameters, denoted as S_1: $\beta = 0.2$, $\gamma = 0.6$, and $\eta = 0.2$; S_2: $\beta = 0.3$, $\gamma = 0.5$, and $\eta = 0.2$; S_3: $\beta = 0.4$, $\gamma = 0.4$, and $\eta = 0.2$; S_4: $\beta = 0.5$, $\gamma = 0.2$, and $\eta = 0.3$; S_5: $\beta = 0.4$, $\gamma = 0.3$, and $\eta = 0.3$; and S_6: $\beta = 0.3$, $\gamma = 0.3$, and $\eta = 0.4$, were evaluated, with S_5 corresponding to the default setting used in our experiments. The results are shown in Table 9. The values of metric Q are 0.233, 0.228, 0.225, 0.231, 0.227, and 0.236, respectively. These results demonstrate that the metric Q remains relatively stable across different parameter combinations, with the values ranging from 0.225 to 0.236. Although S_5 does not achieve the lowest Q, its result is comparable to

those obtained under the other settings, indicating that MSGP is relatively robust to changes in the weighting factors.

Table 8. The partition quality under different τ values.

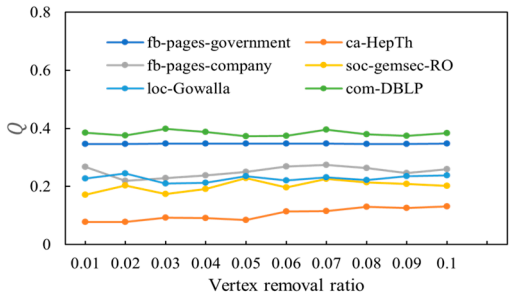
	$\tau = 0.2$		$\tau = 0.4$		$\tau = 0.6$		$\tau = 0.8$		$\tau = 1.0$	
	NSD	***Q***	***NSD***	***Q***	***NSD***	***Q***	***NSD***	***Q***	***NSD***	***Q***
fb-pages-government	0.345	0.364	0.196	0.276	0.182	0.227	0.181	0.232	0.200	0.260
ca-HepTh	0.210	0.258	0.170	0.204	0.140	0.161	0.139	0.194	0.172	0.206
fb-pages-company	0.140	0.165	0.130	0.145	0.122	0.143	0.123	0.147	0.138	0.156
soc-gemsec-RO	0.307	0.335	0.162	0.221	0.154	0.189	0.157	0.215	0.274	0.327
loc-Gowalla	0.167	0.197	0.232	0.240	0.198	0.222	0.202	0.242	0.239	0.250
com-DBLP	0.235	0.221	0.335	0.203	0.150	0.184	0.135	0.187	0.169	0.196

Table 9. The partition quality under different β, γ, and η.

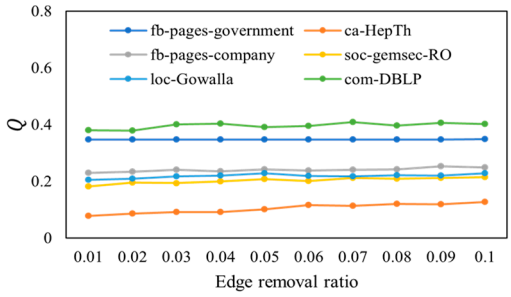
	$\boldsymbol{\beta}$	$\boldsymbol{\gamma}$	$\boldsymbol{\eta}$	Q
S_1	0.2	0.6	0.2	0.233
S_2	0.3	0.5	0.2	0.228
S_3	0.4	0.4	0.2	0.225
S_4	0.5	0.2	0.3	0.231
S_5(default)	0.4	0.3	0.3	0.227
S_6	0.3	0.3	0.4	0.236

Experiment 4. *Stability Analysis.*

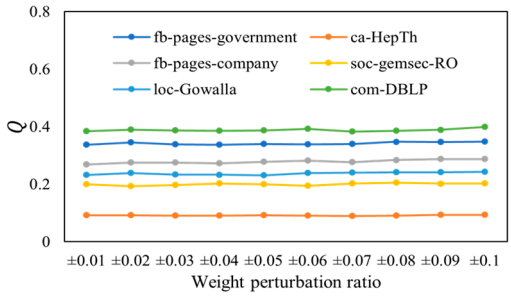
To evaluate the robustness of MSGP, we conducted stability experiments under three types of graph perturbations: (a) vertex removal, where 1%, 2%, 3%, ..., 10% of vertices are randomly selected and removed, and the induced subgraph of the remaining vertices is used as the perturbed graph; (b) edge removal, where 1%, 2%, 3%, ..., 10% of edges are randomly selected and removed; and (c) weight perturbation, where the weights of vertices and edges are randomly perturbed by $\pm$1%, $\pm$2%, ..., $\pm$10%. The resulting perturbed graphs were saved and subsequently used as fixed input graphs for the partitioning experiments. All experiments were performed on all datasets with a fixed block count $k = 32$. For each perturbation setting, MSGP was reapplied to the corresponding perturbed graph, and the partition quality metric Q was computed. The results are illustrated in Figure 7. As the perturbation ratio increases, the value of Q remains generally stable across all three types of perturbations, indicating that the partition quality of MSGP is relatively insensitive to the tested graph perturbations. Compared with edge removal and weight perturbation, vertex removal introduces relatively larger fluctuations in partition quality. This is because removing a vertex simultaneously eliminates all its incident edges, resulting in more substantial changes to the graph topology and connectivity. In contrast, edge removal only directly affects the corresponding local connectivity, while weight perturbation preserves the graph structure, leading to comparatively milder changes in partition quality.



(**a**) Graph partition quality under vertex removal



(**b**) Graph partition quality under edge removal



(**c**) Graph partition quality under weight perturbation

Figure 7. Stability under graph perturbations.

6. Conclusions

Unlike traditional graph partitioning approaches that treat all blocks uniformly, we propose a master–slave graph partitioning paradigm to improve the efficiency of parallel and distributed graph processing through structured inter-block communication. The key idea is to align the partitioning result with the master–slave computing model, in which a single master block is responsible for coordinating inter-block dependencies, while multiple slave blocks remain mutually independent and can be processed concurrently. By organizing inter-block communication into a structured, master-centric pattern, the proposed approach simplifies communication topology and reduces coordination overhead. To realize this paradigm, a two-stage partitioning framework is developed. In the initial stage, inspired by the *k*-VCP technique, a scoring function is designed to estimate the priority of each vertex for inclusion in the master block, thereby guiding the construction of an initial partition that conforms to the master–slave structure. In the refinement stage, the partition is further optimized through a master minimization strategy and a slave balancing strategy, which together reduce the master load while maintaining balanced workloads across slave blocks. Experimental results on six real-world datasets demonstrate that MSGP consistently achieves the best overall partition quality metric Q, with the normalized standard deviation of slave loads (NSD) ranging from 0.03 to 0.30 and the maximum deviation ratio (DEV) below 0.68, substantially outperforming Metis, FLS, LocalFlow, and CNC, which generally exhibit substantially higher NSD and DEV values. The ablation study shows that the two-stage refinement reduces Q from 0.521 (initial partition) to 0.227 (fully optimized), a 56%

improvement, confirming the complementary contribution of master reduction and slave rebalancing. On the com-DBLP dataset, MSGP consistently achieves the lowest PageRank execution time among all compared partitioning methods, which aligns with its optimal Q metric performance under our experimental settings.

Despite these advantages, the proposed approach has several limitations. First, the effectiveness of the master–slave structure relies on the existence of a relatively sparse inter-block connectivity pattern. For graphs where the feasibility conditions are not satisfied, such as dense and highly interconnected graphs with broadly distributed dependencies, no valid master–slave partition exists by definition, and the initial partition step cannot construct a star-shaped topology. Extending the method to such graphs would require a fundamentally different partitioning objective beyond the current problem formulation. Second, the current work focuses on the partitioning stage prior to execution and is designed to produce partitions compatible with the master–slave programming model, rather than optimizing the runtime execution process itself. Third, the proposed method is primarily tailored to static graphs and specific classes of algorithms, and its applicability to dynamic graphs or evolving workloads remains an open problem. Fourth, the experiments are conducted in an environment where multiple logical blocks are co-located on physical machines, with intra-machine communication mediated through shared-memory buffers. Consequently, network latency, bandwidth contention, and cross-machine communication delays are not explicitly captured. Future work should evaluate MSGP in a truly distributed environment to assess its performance under realistic network communication conditions. In addition, the interaction between partitioning strategies and underlying system architectures has not been fully explored. Future work will focus on dynamic graphs, evolving workloads, realistic distributed environments, and system-aware partitioning.

Author Contributions: Conceptualization, G.Z. and T.C.; Methodology, T.C.; Software, G.Z.; Formal Analysis, T.C. and Y.J.; Investigation, T.C.; Resources, G.Z.; Data Curation, Y.J.; Writing—Original Draft, T.C.; Writing—Review and Editing, G.Z. and T.C.; Visualization, T.C. and Y.J.; Funding Acquisition, G.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Natural Science Foundation of China (Grant No. 62072337) and the National Key Research and Development Program of China (Grant No. 2019YFB1704100).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are openly available in SNAP at https://snap.stanford.edu/data/ (accessed on 1 July 2025). The source code and experimental scripts used in this study can be provided by the authors upon reasonable request.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Rahimian, F.; Payberah, A.H.; Girdzijauskas, S.; Jelasity, M.; Haridi, S. Ja-Be-Ja: A distributed algorithm for balanced graph partitioning. In Proceedings of the IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems, Philadelphia, PA, USA, 9–13 September 2013; pp. 51–60.
2. Stanton, I.; Kliot, G. Streaming graph partitioning for large distributed graphs. In Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Beijing, China, 12–16 August 2012; pp. 1222–1230.
3. Gottesbüren, L.; Heuer, T.; Maas, N.; Sanders, P.; Schlag, S. Scalable high-quality hypergraph partitioning. *ACM Trans. Algorithms* **2024**, *20*, 9. [CrossRef]
4. Hendrickson, B.; Leland, R.W. A multi-level algorithm for partitioning graphs. In Proceedings of the ACM/IEEE Conference on Supercomputing, San Diego, CA, USA, 8 December 1995; pp. 1–14.

5. Malewicz, G.; Austern, M.H.; Bik, A.J.; Dehnert, J.C.; Horn, I.; Leiser, N.; Czajkowski, G. Pregel: A system for large-scale graph processing. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, Indianapolis, IN, USA, 6–10 June 2010; pp. 135–146.
6. Tian, Y.; Balmin, A.; Corsten, S.A.; Tatikonda, S.; McPherson, J. From 'think like a vertex' to 'think like a graph'. *Proc. VLDB Endow.* **2013**, *7*, 193–204. [CrossRef]
7. Xin, R.S.; Gonzalez, J.E.; Franklin, M.J.; Stoica, I. GraphX: A resilient distributed graph system on Spark. In Proceedings of the First International Workshop on Graph Data Management Experiences and Systems, New York, NY, USA, 23 June 2013; pp. 1–6.
8. Gan, X. TianheGraph: Topology-aware graph processing. *ACM Trans. Archit. Code Optim.* **2025**, *22*, 112. [CrossRef]
9. Gan, X.; Wu, G.; Qiu, S.; Xiong, F.; Si, J.; Fang, J.; Dong, D.; Gong, C.; Li, T.; Wang, Z. GraphCube: Interconnection hierarchy-aware graph processing. In Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, Edinburgh, UK, 2–6 March 2024; pp. 160–174.
10. Gan, X.; Zhang, Y.; Zeng, R.; Liu, J.; Wang, R.; Li, T.; Chen, L.; Lu, K. Xtree: Traversal-based partitioning for extreme-scale graph processing on supercomputers. In Proceedings of the IEEE 38th International Conference on Data Engineering, Virtual, 9–12 May 2022; pp. 2046–2059.
11. Fey, M.; Lenssen, J.E. Fast graph representation learning with PyTorch Geometric. *arXiv* **2019**, arXiv:1903.02428.
12. Kernighan, B.W.; Lin, S. An efficient heuristic procedure for partitioning graphs. *Bell Syst. Tech. J.* **1970**, *49*, 291–307. [CrossRef]
13. Fiduccia, C.M.; Mattheyses, R.M. A linear-time heuristic for improving network partitions. In *Twenty-Five Years of Electronic Design Automation*; Association for Computing Machinery: New York, NY, USA, 1988; pp. 241–247.
14. Devine, K.; Boman, E.; Heaphy, R.; Hendrickson, B.; Vaughan, C. Zoltan data management services for parallel dynamic applications. *Comput. Sci. Eng.* **2002**, *4*, 90–96. [CrossRef]
15. Zeng, L.; Huang, H.; Zheng, B.; Yang, K.; Shao, S.; Zhou, J.; Xie, J.; Zhao, R.; Chen, X. WindGP: Efficient Graph Partitioning on Heterogeneous Machines. *Proc. VLDB Endow.* **2024**, *14*, 1–15.
16. Shen, J.; Varbanescu, A.L.; Lu, Y.; Zou, P.; Sips, H. Work-load partitioning for accelerating applications on heterogeneous platforms. *IEEE Trans. Parallel Distrib. Syst.* **2015**, *27*, 2766–2780. [CrossRef]
17. Zhang, F.; Zhai, J.; Wu, B.; He, B.; Chen, W.; Du, X. Automatic irregularity-aware fine-grained workload partitioning on integrated architectures. *IEEE Trans. Knowl. Data Eng.* **2019**, *33*, 867–881. [CrossRef]
18. Chevalier, C.; Pellegrini, F. PT-Scotch: A tool for efficient parallel graph ordering. *Parallel Comput.* **2008**, *34*, 318–331. [CrossRef]
19. Tsourakakis, C.; Gkantsidis, C.; Radunovic, B.; Vojnovic, M. Fennel: Streaming graph partitioning for massive scale graphs. In Proceedings of the 7th ACM International Conference on Web Search and Data Mining, New York, NY, USA, 24–28 February 2014; pp. 333–342.
20. Chen, R.; Shi, J.; Chen, Y.; Zang, B.; Guan, H.; Chen, H. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Trans. Parallel Comput.* **2019**, *5*, 13.
21. Ding, Z.; Xiang, Y.; Wang, S.; Xie, X.; Zhou, S.K. Play like a vertex: A Stackelberg game approach for streaming graph partitioning. *Proc. ACM Manag. Data* **2024**, *2*, 162. [CrossRef]
22. Zhou, A.C.; Shen, B.; Xiao, Y.; Ibrahim, S.; He, B. Cost-aware partitioning for efficient large graph processing in geo-distributed datacenters. *IEEE Trans. Parallel Distrib. Syst.* **2019**, *31*, 1707–1723. [CrossRef]
23. Mayer, C.; Tariq, M.A.; Mayer, R.; Rothermel, K. Graph: Traffic-aware graph processing. *IEEE Trans. Parallel Distrib. Syst.* **2018**, *29*, 1289–1302. [CrossRef]
24. Walshaw, C.; Cross, M. Multilevel mesh partitioning for heterogeneous communication networks. *Future Gener. Comput. Syst.* **2001**, *17*, 601–623. [CrossRef]
25. Huang, S.; Aubanel, E.; Bhavsar, V.C. PaGrid: A mesh partitioner for computational grids. *J. Grid Comput.* **2006**, *4*, 71–88. [CrossRef]
26. Dehnashi, M.N.; Momtazpour, M.; Javadi, S.A. A communication-aware scheduler for containers in a Kubernetes environment using Girvan-Newman clustering. In Proceedings of the 32nd International Conference on Electrical Engineering, Tehran, Iran, 14–16 May 2024; pp. 1–5.
27. Blondel, V.; Guillaume, J.L.; Lambiotte, R. Fast unfolding of communities in large networks: 15 years later. *J. Stat. Mech. Theory Exp.* **2024**, *2024*, 10R001. [CrossRef]
28. Onizuka, M.; Fujimori, T.; Shiokawa, H. Graph partitioning for distributed graph processing. *Data Sci. Eng.* **2017**, *2*, 94–105. [CrossRef]
29. Delling, D.; Goldberg, A.V.; Werneck, R.F. Faster batched shortest paths in road networks. In Proceedings of the 11th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems, Saarbrücken, Germany, 8 September 2011; pp. 52–63.
30. Zhang, Y.; Yang, M.; Baghdadi, R.; Kamil, S.; Shun, J.; Amarasinghe, S. GraphIt: A high-performance DSL for graph analytics. *arXiv* **2018**, arXiv:1805.00923.
31. Delimitrou, C.; Kozyrakis, C. Paragon: QoS-aware scheduling for heterogeneous datacenters. *ACM SIGPLAN Not.* **2013**, *48*, 77–88.

32. Zheng, A.; Labrinidis, A.; Chrysanthis, P.K.; Lange, J. Argo: Architecture-aware graph partitioning. In Proceedings of the IEEE International Conference on Big Data, Washington, DC, USA, 5–8 December 2016; pp. 284–293.
33. Jafari, N.; Selvitopi, O.; Aykanat, C. Fast shared-memory streaming multilevel graph partitioning. *J. Parallel Distrib. Comput.* **2021**, *147*, 140–151. [CrossRef]
34. Zhou, Y.; Wang, G.; Zhou, M. Detecting k-vertex cuts in sparse networks via a fast local search approach. *IEEE Trans. Comput. Soc. Syst.* **2023**, *11*, 1832–1841. [CrossRef]
35. Parter, M.; Petruschka, A. Near-optimal distributed computation of small vertex cuts. *Distrib. Comput.* **2024**, *37*, 67–88. [CrossRef]
36. Cornaz, D.; Furini, F.; Lacroix, M.; Malaguti, E.; Mahjoub, A.R.; Martin, S. The vertex k-cut problem. *Discret. Optim.* **2019**, *31*, 8–28. [CrossRef]
37. Nanongkai, D.; Saranurak, T.; Yingchareonthawornchai, S. Breaking quadratic time for small vertex connectivity and an approximation scheme. In Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, Phoenix, AZ, USA, 23–26 June 2019; pp. 241–252.
38. Jiang, Y.; Mukhopadhyay, S. Finding a small vertex cut on distributed networks. In Proceedings of the 55th Annual ACM Symposium on Theory of Computing, Orlando, FL, USA, 20–23 June 2023; pp. 1791–1801.
39. Fan, W.; Jin, R.; Liu, M.; Lu, P.; Luo, X.; Xu, R.; Yin, Q.; Yu, W.; Zhou, J. Application driven graph partitioning. In Proceedings of the ACM SIGMOD International Conference on Management of Data, Online, 14–19 June 2020; pp. 1765–1779.
40. Diestel, R. *Graph Theory*; Springer Nature: Berlin/Heidelberg, Germany, 2025.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.