NOVEL MODELS AND METHODS FOR ACCELERATING PARALLEL FULL-BATCH GNN TRAINING ON DISTRIBUTED-MEMORY SYSTEMS

A THESIS SUBMITTED TO

THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER ENGINEERING

By

Ahmet Can Bağırgan

July 2025

Novel Models and Methods for Accelerating Parallel Full-Batch GNN Training on Distributed-Memory Systems
By Ahmet Can Bağırgan
July 2025

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Cevdet Aykanat(Advisor)

Hamdi Dibeklioğlu

Murat Manguoğlu

Approved for the Graduate School of Engineering and Science:

Orhan Arıkan
Director of the Graduate School

ABSTRACT

NOVEL MODELS AND METHODS FOR ACCELERATING PARALLEL FULL-BATCH GNN TRAINING ON DISTRIBUTED-MEMORY SYSTEMS

Ahmet Can Bağırgan
M.S. in Computer Engineering
Advisor: Cevdet Aykanat
July 2025

Graph Neural Networks (GNNs) have emerged as effective tools for learning from graph-structured data across diverse application domains. Despite their success, the scalability of GNNs remains a critical challenge, particularly in full-batch training on large-scale, irregularly sparse, and scale-free graphs. Traditional one-dimensional (1D) vertex-parallel training strategies, while widely adopted, often suffer from severe load imbalance and excessive communication overhead, limiting their performance on distributed-memory systems. This thesis addresses the scalability limitations of 1D approaches by investigating alternative partitioning strategies for parallelization that better exploit the structure of modern graph workloads. A systematic evaluation framework is developed to assess parallel GNN training performance across a range of datasets with varying sparsity and degree distributions. The framework captures key performance indicators such as computational load balance, inter-process communication volume, and parallel runtime. Extensive experiments are conducted on two Tier-0 supercomputers, LUMI and MareNostrum5, using hundreds of real-world graph instances. On average of 22 well-known GNN datasets, the results show up to 61% decrease in total communication volume and up to 39% decrease in parallel runtime compared to 1D partitioning strategies on 1024 processes. These improvements are consistent across graphs with high variance in degree and sparsity, confirming the robustness of the proposed approaches. The findings demonstrate the potential of moving beyond traditional 1D paradigms and provide practical insights into scalable and communication-efficient GNN training on distributed platforms.

Keywords: Graph neural networks, graph partitioning, parallel computing, distributed systems.

ÖZET

DAĞITIK-BELLEKLİ SİSTEMLERDE PARALEL ÇİZGE SİNİR AĞLARI EĞİTİMİNİ HIZLANDIRMAK İÇİN YENİ MODEL VE YÖNTEMLER

Ahmet Can Bağırgan
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Cevdet Aykanat
Temmuz 2025

Çizge Sinir Ağları (GNN'ler), çizge tabanlı verilerden öğrenme konusunda güçlü bir araç olarak öne çıkmaktadır. Ancak, özellikle büyük ölçekli, düzensiz seyrek ve ölçeklenemeyen çizgelerde tam yığın (full-batch) eğitim senaryolarında, bu ağların dağıtık sistemlerdeki ölçeklenebilirliği ciddi zorluklar içermektedir. Yaygın olarak kullanılan tek boyutlu (1D) düğüm-paralel stratejiler, hesaplama yükü dengesizliği ve yüksek iletişim maliyetleri nedeniyle sınırlı verimlilik sunmaktadır. Bu tez, modern çizge verilerinin yapısal özelliklerini daha iyi değerlendirebilen alternatif bölütleme stratejileri ile 1D yaklaşımların ölçeklenebilirlik sınırlamalarını ele almaktadır. Bu kapsamda, farklı seyreklik ve derece dağılımlarına sahip veri kümeleri üzerinde paralel GNN eğitiminin performansını değerlendiren sistematik bir çerçeve geliştirilmiştir. Çalışmada, hesaplama yük dengesi, süreçler arası iletişim hacmi ve paralel çalışma süresi gibi temel performans ölçütleri dikkate alınmıştır. LUMI ve MareNostrum5 gibi Tier-0 süper bilgisayarlarda, yüzlerce gerçek çizge üzerinde yapılan kapsamlı deneylerde, 22 yaygın kullanılan GNN veri kümesi ortalamasına göre 1D bölütlemeye kıyasla %61'e varan iletişim hacmi ve %39'a varan paralel çalışma süresi azalmaları elde edilmiştir. Bu kazanımlar, önerilen yaklaşımların derece ve seyreklik açısından değişken yapıda çizgeler üzerinde dahi tutarlı ve etkili olduğunu göstermekte; dağıtık sistemlerde ölçeklenebilir ve iletişim açısından verimli GNN eğitimi için değerli katkılar sunmaktadır.

Anahtar sözcükler: Çizge sinir ağları, çizge bölümleme, parallel hesaplama, dağıtık sistemler.

Acknowledgement

I would like to express my gratitude to my supervisor, Prof. Dr. Cevdet Aykanat for guidance, suggestions, and encouragement throughout the development of this thesis.

I am grateful to Prof. Dr. Murat Manguoğlu and Asst. Prof. Dr. Hamdi Dibeklioğlu for reading and commenting on the thesis.

I am grateful to all of my friends for their moral and intellectual support during my studies, especially to Alp, Kübra, and Yahya.

I would like to thank my fellow teammates, Ahmet, Erkin, Kutay, and Serdar, for the enjoyable time we shared during our collaboration.

I owe special thanks to dear İmran and my little brother Kaan for their support and enjoyable hospitality.

And finally, I would like to express my deepest gratitude to my mom and dad, who have supported me unconditionally and whose love I have always felt.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

For over three decades, one-dimensional (1D) vertex-parallel codes have been widely developed and effectively used for solving irregularly sparse graph problems on large-scale distributed-memory systems. These methods have become standard practice, forming the basis of many legacy codes. The popularity of 1D vertex-parallel approaches stems from their reliance on low-dimensional partitioning of graph-theoretical computation domains, which enables them to maintain coherence along a single dimension and allows relatively simple and efficient parallel implementations. Their success is largely due to the availability of robust and scalable graph and hypergraph partitioning tools, as well as high-performance parallel implementations of these tools [1, 2, 3, 4, 5, 6, 7, 8]. In particular, support for multi-constraint partitioning in these tools allows balancing multiple computational loads simultaneously, which is essential in multi-stage parallel applications.

However, the increase of highly scale-free graphs in modern applications presents serious challenges to traditional 1D graph and hypergraph partitioning models. These graphs typically contain a small number of extremely high-degree vertices, which significantly complicates the task of balancing computational and communication loads across processes. As a result, the quality of the partitions

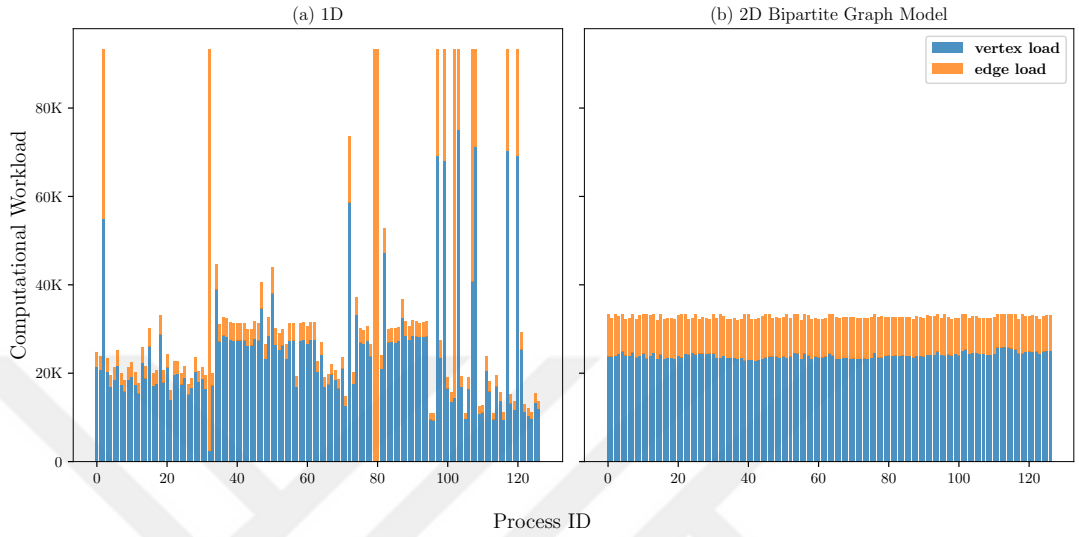


Figure 1.1: Computational load distribution attained by (a) 1D vertex partitioning, (b) 2D edge-vertex-based partitioning using proposed bipartite graph model for `boyd1` on 128 processes

generated by 1D models deteriorates, especially in terms of cutsize, which corresponds to the total volume of inter-process communication.

To address this issue, two-dimensional (2D) edge-based partitioning models and 2D edge-vertex-parallel algorithms have recently gained attention as a promising alternative to traditional 1D approaches [9, 10, 11, 12]. By increasing the degrees of freedom in the partitioning process, 2D edge-based methods ease the constraints on load balancing and enable the partitioning tool to produce higher-quality results, especially for scale-free and irregularly sparse graphs.

Figure 1.1 illustrates this with an example on a 128-way partition of the highly scale-free grapzh `boyd1`, which has 93K vertzices and 1.2M edges, and a coefficient of variation of value of 50.3 on vertex degrees. In this evaluation, edges and vertices are assigned computational weights of 2 and $k + 1 = 33$, respectively, where k is the embedding size. Figure 1.1(a) shows the load distribution resulting from a 1D vertex partitioning using the conventional graph model, where each vertex is weighted by its degree plus k. Although the current graph partitioning tools [1, 2, 3, 4] are known for producing high-quality balanced partitions, the figure reveals that the 1D model suffers from severe computational imbalance:

12 of the 128 processes are assigned workloads exceeding 93K (almost triple the average load of 33K) due to just 18 high-degree vertices with degrees ranging from 3.7K to 93K.

In contrast, 2D edge-vertex-parallel computations require simultaneous balancing of both edge and vertex loads. This is particularly important in applications such as Graph Convolutional Networks (GCNs), where each iteration consists of edge-based computations followed by vertex updates. If the synchronization between these two phases is tight, the partitioning problem can be modeled as a two-constraint balancing problem. If not, a single-constraint formulation with properly chosen weights for edge and vertex tasks can still be sufficient. However, to the best of our knowledge, the existing literature lacks 2D partitioning models that explicitly support the simultaneous balancing of edge and vertex computation loads.

In this thesis, we propose 2D edge-based graph partitioning models and methods designed to achieve such simultaneous balancing. Our main contribution is a bipartite graph model that represents the original graph using two types of nodes: one set for edges (edge-nodes) and another for vertices (vertex-nodes). This structure allows assigning different weights to edge- and vertex-nodes, enabling fine-grained control over the balance of edge and vertex computations during partitioning.

Figure 1.1(b) demonstrates the improved balance achieved using our bipartite graph model on the same graph instance, `boyd1`. Here, edge-nodes and vertex-nodes are weighted with values of 2 and $k+1$, respectively. The resulting computational load distribution is significantly more balanced, with imbalance reduced from 291% in the 1D case to just 2%. Moreover, the total communication volume is also drastically reduced: the 2D model achieves a cutsize of only 19K words compared to 521K words in the 1D vertex partitioning.

The main drawback of the 2D edge-based partitioning approach is its increased memory requirement during partitioning [13]. In a graph with $|V|$ vertices and $|E|$ edges, our bipartite model creates $|V|+|E|$ nodes and $2|E|$ edges, leading to

higher memory usage compared to traditional 1D models. To overcome this, we introduce a simple yet effective edge-clustering method that significantly reduces the memory footprint of the 2D model. Applied to 22 benchmark graphs, our method reduces the number of nodes and edges in the bipartite graph to an average of $1.55|V|$ and $1.14|E|$, respectively. Importantly, this reduction preserves the computational balance benefits of the original model and even improves communication efficiency, yielding an average 20% reduction in total communication volume for 1024-way partitioning.

In our bipartite graph model, computational load balancing is encoded explicitly through edge-nodes and vertex-nodes. The balancing of vertex-nodes is also indirectly related to communication volume balance, as vertex replication is a primary source of communication overhead in parallel graph computations. To further enhance communication balance, we develop a P-feasible bin-packing-based heuristic to assign vertex-nodes to processes where P is the number of processes.. This heuristic does not affect the total or average communication volume but reduces the peak communication volume across processes, thereby making communication load metrics more fair and informative.

The remainder of the thesis is organized as follows: Chapter 2 provides background on Graph Neural Networks (GNNs) and graph partitioning, covering the foundational concepts relevant to our work. Chapter 3 presents a review of related literature on parallel GNN training and partitioning strategies. In Chapter 4, we describe our application framework and introduce a generalized 2D parallelization scheme for graph computations, along with the partitioning notation we use. Chapter 5 details the proposed partitioning models and methods for jointly balancing edge and vertex computation loads. Chapter 6 provides implementation details, datasets, and evaluation metrics. Chapter 7 presents the experimental results. Finally, Chapter 8 concludes the thesis with a summary of findings and potential directions for future work.

Chapter 2

Background

2.1 Graph Partitioning

Consider a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ that consists of a set of vertices $\mathcal{V}$ and a set of edges $\mathcal{E}$. Each edge $(v_i, v_j) \in \mathcal{E}$ connects exactly two vertices. Each vertex v_i can be associated with a weight $w(v_i)$, and each edge (v_i, v_j) can be associated with a cost $w((v_i, v_j))$. $\Pi(\mathcal{G}) = \{\mathcal{V}_1, \mathcal{V}_2, \mathcal{V}_3, ..., \mathcal{V}_K\}$ is said to be a K-way partition of $\mathcal{G}$ if parts are mutually disjoint ($\mathcal{V}_x \cap \mathcal{V}_y = \emptyset$ for each pair of parts $\mathcal{V}_x$ and $\mathcal{V}_y$) and mutually exhaustive ($\mathcal{V}_x \cup \mathcal{V}_y \ ... \cup \ \mathcal{V}_k = \mathcal{V}$).

In a partition Π, an edge is said to be cut if v_i and v_j are in different parts. The cut size of a partition Π is defined as the sum of the costs of the cut edges, that is

$$cutsize(\Pi) = \sum_{(v_i,v_j)\in\mathcal{E}_{cut}} \omega((v_i, v_j)). \tag{2.1}$$

In a partition Π, the weight/load of a part W_k is defined as the sum of the weights of its vertices, i.e., ($W_k = \sum_{v_i\in\mathcal{V}_k} \omega(v_i)$). Partition Π is said to be balanced if each part $\mathcal{V}_k$ satisfies $\omega(\mathcal{V}_k) \leq (1 + \epsilon) \cdot W_{avg}$, where ϵ is the maximum allowable imbalance ratio and W_{avg} is the average part weight ($\sum_k W_k)/K$). In the graph

partitioning problem, the partitioning constraint is to satisfy the balance constraints among the part weights, whereas the partitioning objective is to minimize the cut size. The graph partitioning problem is known to be *NP*-Hard [14, 15]. So, several heuristic-based multilevel algorithms [16, 17, 1] have been proposed to partition a graph, which led to successful multilevel tools such as Metis [1], KaHIP [3], Scotch [2], and Chaco [4]. A detailed explanation of the multilevel graph partitioning framework, including its coarsening, initial partitioning, and uncoarsening phases, is provided in Appendix A.

2.2 Graph Neural Networks

Graph Neural Networks (GNNs) are a class of neural networks designed to operate on graph-structured data. Unlike traditional neural networks that assume regular input formats (like grids in images), GNNs can directly model relationships and interactions between entities represented as vertices and edges. They work by iteratively aggregating and transforming information from a node's neighbors to learn expressive vertex, edge, or graph-level representations. This makes GNNs particularly powerful for tasks such as vertex classification, link prediction, and graph classification. They are mostly used in applications such as social networks, recommendation systems, molecular chemistry, and knowledge graphs.

While a wide variety of GNN architectures/models [18, 19, 20] have been proposed in the literature, this study targets Graph Convolutional Networks (GCNs) as an application in the context of the vertex classification problem. A GCN layer consists of two operations: aggregation and update. To formalize these steps, Besta and Hoefler [21] introduced two alternative perspectives: the local formulation, which is grounded in vertex-theoretic principles, and the global formulation, which expresses these computations using matrix multiplications.

2.2.1 Local Formulation

The local formulation, also known as the message passing view, expresses GNN computation in terms of operations over individual vertices and their immediate neighborhoods. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be an undirected graph, where $\mathcal{V}$ is the set of vertices and $\mathcal{E}$ is the set of edges. Each vertex $v_i \in \mathcal{V}$ is associated with a feature vector $h_i^{(\ell)} \in \mathbb{R}^{d_\ell}$ at layer ℓ. The local formulation computes the feature vector at the next layer as:

$$h_i^{(\ell+1)} = \phi\left(h_i^{(\ell)}, \bigoplus_{j \in \mathcal{N}(i)} \psi\left(h_i^{(\ell)}, h_j^{(\ell)}\right)\right), \tag{2.2}$$

where $\mathcal{N}(i)$ denotes the set of neighboring vertices of i. Here, ψ acts on each neighbor j's feature vector by multiplying it with a scalar $1/\sqrt{d_i d_j}$ for normalization. $\bigoplus$ is a permutation invariant aggregation function such as sum, max or average. And in each layer, an activation function ϕ is used for transformation.

Each layer ℓ has two main operations: a permutation-invariant aggregation function $\bigoplus$ (such as sum, mean, or max) and a learnable linear transformation function ϕ. The aggregation function $\bigoplus$ gathers feature vectors from a vertex and its 1-hop neighbors to encode local structural context, while the transformation function ϕ, parameterized by a weight matrix $W^{(\ell)}$ of shape $f \times k$, maps the aggregated features into a new representation space.

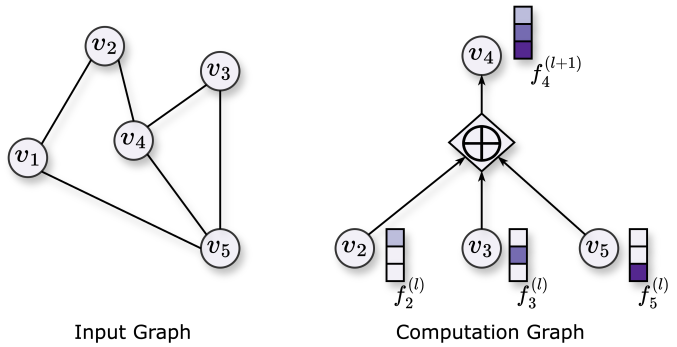


Figure 2.1: Example of message passing scheme

Figure 2.1 clarifies the message passing concept in a vertex-centric (local) fashion. To obtain a hidden vector $f_4^{(\ell+1)}$ of v_4, $f_2^{(\ell)}$, $f_3^{(\ell)}$ and $f_5^{(\ell)}$ are aggregated using

a summation function. This obtained vector is then combined with v_4's own feature vector $f_4^{(\ell)}$, and the result is linearly projected, followed by a non-linear activation, typically ReLU, to produce the updated version. This process reflects the general update rule in the local formulation, where each vertex updates its embedding based on information received from its immediate neighbors.

2.2.2 Global Formulation

The global formulation, alternatively referred to as the matrix-based view, encodes GNN operations using linear algebraic expressions over the entire graph. Here, vertex features at layer ℓ are stored in a matrix $H^{(\ell)} \in \mathbb{R}^{n \times d_{ell}}$, where each row corresponds to a vertex embedding. Message passing is performed via multiplication with a normalized adjacency matrix $\hat{A} \in \mathbb{R}^{n \times n}$, leading to the canonical GCN update:

$$Z^{(\ell)} = \hat{A} \cdot H^{(\ell)} \tag{2.3}$$

$$H^{(\ell+1)} = \sigma(Z^{(\ell)} \cdot W^{(\ell)}) \tag{2.4}$$

where $W^{(\ell)} \in \mathbb{R}^{d_\ell \times d_{\ell+1}}$ is a trainable weight matrix and σ is an activation function, typically ReLU. Adjacency matrix A is normalized with $1/\sqrt{d_i d_j}$ coefficient (mentioned in local formulation) using $\hat{A} = \tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}}$. (further details on this normalization can be found in [18]).

Using this formulation, a GCN layer can be represented as a chain of matrix multiplications. The operation in equation 2.3 corresponds to a sparse-dense matrix multiplication (SpMM), whereas the operation in equation 2.4 involves the multiplication of two dense matrices, thus it corresponds to a generalized matrix multiplication (GeMM). These are very well-known and commonly used kernels in high performance computing field.

This algebraic abstraction enables highly optimized, vectorized implementations that leverage SpMM and GeMM kernels. Libraries such as DGL [22], PyTorch Geometric [23], and TensorFlow Graph [24] use the global formulation

internally to achieve scalable performance on large datasets.

Since the global formulation is adopted in this study, the remainder of the thesis will proceed using only the matrix-view notation and corresponding formulations.

2.3 Computational Loads of SpMM/GeMM Kernels in Distributed Memory Data Parallelism

In distributed memory systems, large-scale matrix computations are often parallelized by partitioning the input matrices across multiple processes. Although the abovementioned kernels (SpMM and GeMM) both involve matrix operations, the underlying data structures and computational patterns differ significantly, which has direct implications on how computational load should be balanced in distributed environments.

The SpMM operation is defined as the multiplication of a sparse matrix $A \in \mathbb{R}^{n \times n}$ with a dense matrix $X \in \mathbb{R}^{n \times f}$, producing a dense output matrix $Y = AX \in \mathbb{R}^{n \times f}$. In distributed memory data parallelism, matrix A is typically partitioned by rows, and each process is assigned a subset of the rows along with the corresponding entries of X required for local computation.

The computational load of SpMM is directly determined by the number of nonzero entries in the assigned submatrix. Each nonzero element a_{ij} in row i contributes to f floating point operations, corresponding to the multiplication of a_{ij} with each column of X. Consequently, the total number of floating-point operations in SpMM is proportional to the number of nonzeros in A, and the local workload of a process is proportional to the number of nonzeros in the rows it owns. This makes edge-balanced partitioning strategies, that is, assigning approximately equal number of nonzero elements to each process, critical for

achieving good load balance in parallel SpMM.

The GeMM operation involves the multiplication of two dense matrices. Suppose $H \in \mathbb{R}^{n \times f}$ is multiplied by a weight matrix $W \in \mathbb{R}^{f \times k}$ to produce an output $Z = HW \in \mathbb{R}^{n \times k}$. In distributed settings, the matrix H is typically partitioned row-wise across processes, such that each process is responsible for computing a subset of the output rows.

In contrast to SpMM, the computational load of GeMM is uniform across rows. Each row of H incurs the same number of $(f \times d)$ multiplication-addition operations. As a result, the local computational workload of each process is proportional to the number of rows/vertices assigned to it. Hence, for GeMM, vertex-balanced or row-balanced partitioning is more suitable to ensure that each process performs an equal share of work.

This divergence has important implications for the design of partitioning schemes. A partition that balances vertex counts may yield load imbalance in SpMM, while a partition that balances nonzeros may lead to imbalance in GeMM. Therefore, hybrid or multi-constrained partitioning strategies are often necessary to jointly minimize computational and communication overhead across both kernels. However, since these two kernels are executed consecutively within a GCN layer without requiring global synchronization, a single-constrained partitioning strategy with an appropriate weighting scheme may suffice, rather than employing a multi-constrained approach. This will be discussed in detail in the section where we describe the weighting scheme used for the models proposed in Chapter 5.

2.4 Bipartite Graph

A bipartite graph is a special class of graphs whose vertex set can be partitioned into two disjoint sets such that no edge exists between vertices within the same set. Formally, a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is said to be bipartite if there exists a partition

of $\mathcal{V}$ into two subsets $\mathcal{U}$ and $\mathcal{W}$ (i.e., $\mathcal{V} = \mathcal{U} \cup \mathcal{W}$ and $\mathcal{U} \cap \mathcal{W} = \emptyset$) such that every edge $e = (u, w) \in E$ satisfies $u \in \mathcal{U}$ and $w \in W$. In other words, all edges in the graph connect a vertex from $\mathcal{U}$ to a vertex from $\mathcal{W}$. A bipartite graph is a general structure to model the relationship between two node types. In the following sections, we describe how we can utilize this feature of bipartite graphs in the edge-vertex-balanced partitioning task.

Chapter 3

Related Work

Graph partitioning-based data parallelism is a commonly used preprocessing strategy in GNNs, particularly favored in full-batch training, as it ensures computational load balance across partitions and reduces feature communication overhead [21, 25]. In the literature, partitioning-based GNN parallelism is typically categorized into two main subgroups: 1D vertex partitioning (edge-cut) and 2D edge partitioning (vertex-cut).

Vertex partitioning (edge-cut) offers certain practical advantages in distributed GNN training, which have made it a common choice in early systems such as DistDGL [26] and AliGraph [27]. By assigning disjoint subsets of vertices (along with their incident edges) to different parts/processes, this strategy simplifies memory management and enables efficient computation-to-data locality, since each vertex's feature updates can be computed largely using locally stored information. In DistDGL, for example, the authors leverage vertex partitioning to group feature storage and computation, thereby reducing redundant memory access and improving cache efficiency. Similarly, AliGraph exploits vertex-based sharding to parallelize the training pipeline across large-scale industrial graphs while maintaining acceptable communication overhead. Moreover, vertex partitioning tends to limit vertex replication, which can help contain memory consumption per process. This is especially beneficial in full-batch training, where the entire graph

is processed in each iteration, and minimizing communication across processes becomes crucial for scalability. However, while vertex partitioning provides locality and memory benefits under certain conditions, it is not without limitations, particularly in terms of handling high-degree vertices, which will be discussed in the following.

In contrast to vertex partitioning, 2D edge partitioning assigns edges, rather than vertices, to parts. This approach is adopted in more recent systems such as DistGNN [28] and CAGNET [29], which are designed to scale to billion-scale graphs. One of the primary advantages of edge partitioning is its ability to achieve better load balancing of message-passing computations, as edge-centric operations (e.g., aggregations) are evenly distributed across processes. Additionally, by distributing edges instead of vertices, systems can reduce the number of cross-partition communications per edge, especially when combined with collaborative sampling strategies, as in DistGNN.

Edge partitioning is also naturally more compatible with fine-grained parallelism, enabling better utilization of hardware resources. For example, CAGNET applies a hybrid execution model that decouples computation and communication by leveraging edge-based locality and overlapping message exchanges with local updates. That said, edge partitioning introduces vertex replication as a trade-off: since vertices may be incident to edges assigned to multiple partitions, their features must be duplicated across processes. This replication increases both memory usage and synchronization overhead, especially during backward passes or when vertex features are frequently updated. Thus, while 2D edge partitioning offers improved parallel efficiency and communication balance, it also requires careful design to control replication-related costs.

Overall, while edge partitioning tends to yield better runtime performance in distributed GNN training, particularly by balancing the edge workload and minimizing communication volume, it often incurs higher memory consumption and longer preprocessing time due to its inherently larger model size. One of the core challenges associated with edge partitioning is the vertex replication problem: although it achieves good edge balance, it does not guarantee vertex balance,

leading to uneven memory usage and computational load across workers. This imbalance can become a significant bottleneck, especially in full-graph training, where feature storage and update costs dominate.

To address the scalability limitations of traditional partitioners, recent work has proposed streaming edge partitioning algorithms such as DBH [30] and its variants [31, 32]. These methods assign edges to partitions in a single-pass streaming manner, making them highly efficient and scalable even for web-scale graphs. However, their solution quality, in terms of minimizing replication factor or ensuring vertex balance, is generally inferior to that of offline partitioners. As a result, while streaming partitioners are suitable for scenarios with tight preprocessing budgets or dynamic graphs, they often fall short in delivering the partitioning quality needed for optimal GNN training performance.

Merkel et al. conduct a comprehensive empirical evaluation of graph partitioning strategies for distributed GNN training, comparing vertex, edge, and hybrid methods across diverse datasets and training configurations [33]. Their findings show that while edge partitioning often results in better runtime performance due to improved edge load balance and reduced communication volume, it also incurs substantial memory overhead from vertex replication. More importantly, the study emphasizes that many edge-based approaches overlook vertex balance [34, 35, 36], which leads to uneven memory and compute loads across workers. In particular, their analysis demonstrates a strong correlation between vertex imbalance and memory consumption, revealing that even when edge workloads are evenly distributed, vertex skew can severely hinder scalability. These observations highlight the critical need for partitioning strategies that simultaneously balance both edges and vertices, as optimizing only for edge distribution is insufficient to fully exploit the parallelism potential in GNN training.

Chapter 4

Application Framework

4.1 Graph Application Framework

We are given a graph application represented by an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. Since $\mathcal{G}$ is undirected, $(u_i, u_j) \in \mathcal{E}$ represents the pair of nonzeros a_{ij} and a_{ji} in adjacency matrix $\mathcal{A}$. The adopted task decomposition scheme for parallelization induces two types of atomic tasks, executed iteratively: *edge tasks* and *vertex tasks*. The atomic task associated with an edge (v_i, v_j) performs a computation using the current values of vertices v_i and v_j, producing a pair of intermediate results that respectively contribute to the updates of vertices v_j and v_i. Subsequently, the atomic task associated with a vertex v_i aggregates the results produced by the edge tasks of all incident edges of v_i, and performs an update operation to determine the new value of v_i for the next iteration.

To illustrate this generalization, we will use GCNs as a typical example of such a graph-based application. In GCN, one layer of computation can be divided into two matrix multiplication steps as follows (also discussed in Section 2.2.2 in detail)

$$Z^{\ell} = AH^{\ell} \tag{4.1}$$

$$H^{\ell+1} = Z^{\ell}W^{\ell}, \tag{4.2}$$

where A is the $n \times n$ sparse adjacency matrix of $\mathcal{G}$, H is the $n \times f$ dense matrix that contains horizontally stacked feature vectors for each vertex, W is the $f \times k$ dense weight matrix, and Z is a $n \times f$ dense temporary matrix. It can easily be seen that Equation 4.1 represents an SpMM operation, while Equation 4.2 corresponds to a GeMM operation.

Here, each edge (v_i, v_j) corresponds to two nonzero entries a_{ij} and a_{ji} in the adjacency matrix A, assuming the graph is undirected, which corresponds to a symmetric adjacency matrix. In this context, each atomic task associated with an edge (v_i, v_j) can be defined as two *DAXPY*-like dense matrix operations as follows

$$Z^{\ell}[i,:] \leftarrow H^{\ell}[i,:] + a_{ij}X[j,:] \tag{4.3}$$

$$Z^{\ell}[j,:] \leftarrow H^{\ell}[j,:] + a_{ji}X[i,:], \tag{4.4}$$

These operations, Equations 4.3 and 4.4, can be considered as the *aggregation* phase in the local formulation.

Each atomic task associated with a vertex v_i can be defined as a GeMM operation as follows

$$H^{\ell+1}[i,:] \leftarrow Z^{\ell}[i,:]\, W^{\ell}. \tag{4.5}$$

This operation, Equation 4.5, can be considered as the *update* phase in the local formulation.

To quantify the computational workload of these atomic tasks, we define two cost functions: $eload((v_i, v_j))$ denotes the computational load of the edge task associated with edge (v_i, v_j), and $vload(v_i)$ denotes the computational load of the vertex task associated with vertex v_i. Assuming the feature dimension is f and the output feature dimension is k, we model the computational loads as follows:

$$eload((v_i, v_j)) = 2f, \quad vload(v_i) = f(k+1). \tag{4.6}$$

Here, $eload((v_i, v_j)) = 2f$ accounts for the two *DAXPY*-like operations in Equations 4.3 and 4.4, each involving an f-dimensional vector. Likewise, $vload(v_i) =$

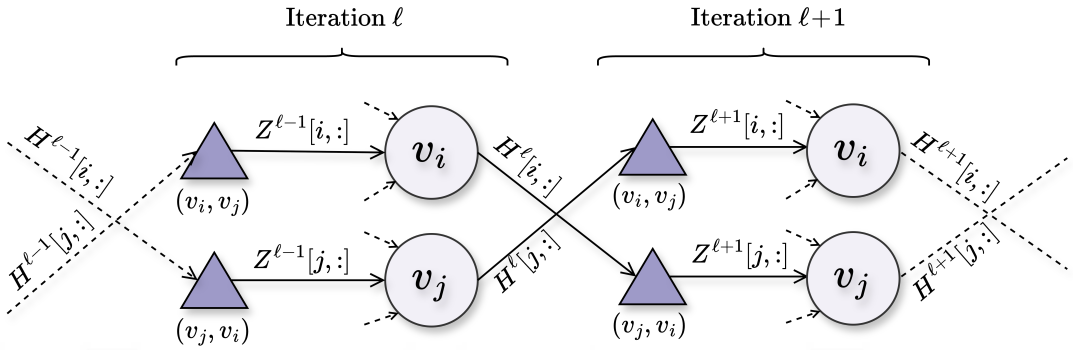


Figure 4.1: Intra-/inter-iteration task/data dependencies/interactions among edge and vertex tasks within the context of two iterations.

$f(k+1)$ arises from the vertex update in Equation 4.5, representing a matrix-vector multiplication ($f \times k$) plus the computation corresponding to self-edge (v_i, v_i).

Figure 4.1 shows the nature of the intra-/inter-iteration task/data dependencies/interactions among these two types of tasks within the context of two successive iterations. In the figure, triangles denote the atomic tasks associated with edges, whereas circles represent atomic tasks associated with vertices.

As seen in Figure 4.1, intra-iteration task dependency occurs from edge tasks to vertex tasks as follows: within each iteration, the execution of the atomic task associated with a vertex v_i depends on the aggregation to be performed on the results of the atomic tasks associated with the incident edges. Inter-iteration task dependency occurs from the vertex task to the edge tasks as follows: between two successive iterations, the result of the atomic task associated with a vertex v_i is scattered to the atomic tasks associated with its incident edges. Intra-iteration task interaction exists since the edges incident to the same vertex contribute to the feature vector of that vertex. Inter-iteration task interaction exists in a dual manner.

4.2 Partitioning

For a given graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, P-way edge-and-vertex-based partition is defined as

$$\Pi(\mathcal{V}, \mathcal{E}) = \{\Pi^1 = (\mathcal{V}^1, \mathcal{E}^1), \Pi^2 = (\mathcal{V}^2, \mathcal{E}^2), ..., \Pi^P = (\mathcal{V}^P, \mathcal{E}^P)\} \tag{4.7}$$

where edge parts as well as vertex parts are mutually disjoint and exhaustive. That is

$$\mathcal{E}^p \cap \mathcal{E}^q = \emptyset \text{ and } \mathcal{V}^p \cap \mathcal{V}^q = \emptyset \tag{4.8}$$

$$\bigcup_q \mathcal{E}^q = \mathcal{E} \text{ and } \bigcup_q \mathcal{V}^q = \mathcal{V} \tag{4.9}$$

for each distinct pair of parts p and q.

In a given partition Π, without loss of generality, we assume that part Π^q is assigned to P^q. We will summarize the interprocess communication requirements for such a 2D partition.

We introduce the notation for edges incident to a vertex and vice versa as follows

$$\delta(v_i) = \{(v_i, v_j) \mid v_j \in Adj(v_i)\} \tag{4.10}$$

Equation 4.10 refers to the set of incident edges to a given vertex.

In Π, a local edge of a part/process is referred to as a boundary edge if at least one of its two incident vertices is owned by another process. Similarly, a local vertex of a process is referred to as a boundary vertex if at least one of its incident edges is owned by another process.

Boundary vertices incur an expand/scatter type of communication. Consider a boundary vertex v_i of P^q, P^q should send the data $data(v_i)$ associated with v_i to those processes other than P^q that own at least one edge incident to v_i. That is P^q will send $data(v_i)$ to $P^{r \neq q}$ if $\delta(v_i) \cap \mathcal{E}^r \neq \emptyset$. So, the boundary vertex v_i will incur a communication volume of $|data(v_i)|(\lambda(v_i) - 1)$, where $\lambda(v_i) = \left|\{P^r : \delta(v_i) \cap \mathcal{E}^r \neq \emptyset\}\right|$ can be defined as *edge-connectivity* of v_i in Π.

Boundary edges incur a fold/reduce type of communication as follows. Consider the maximal subset $\{\delta(v_i) \cap \mathcal{E}^q, v_i \in \mathcal{V}^{r \neq q}\}$ of boundary edges of a process P^q that are incident to the same nonlocal vertex v_i in $P^{r \neq q}$. Then, P^q should perform a local reduce operation on the results of these edges to compute a local intermediate result for v_i, referred to as v_i^q here, and it will send this intermediate result to P^r. Possibly $v_i \in \mathcal{V}^r$ will have some other maximal subset of boundary edges in other processes incident to v_i, hence P^r will receive $\lambda(v_i) - 1$ partial results in total to finalize this reduce operation on v_i.

4.3 Generic Edge-Vertex Parallel Graph Algorithm

To better clarify the combinatorial models and optimization methods discussed in the future sections, we provide a generic edge-vertex-parallel graph computation algorithm.

The algorithm operates under the assumption that both edges and vertices of the input graph are assigned a priori to specific parts or processes. As outlined in Algorithm 1, the procedure comprises two main communication phases (pre-communication and post-communication) interleaved with local computation stages.

In the pre-communication phase, each process initially posts non-blocking receive operations to anticipate incoming messages from remote processes that own the halo vertices required for computations involving its local boundary edges. A vertex or edge is classified as a halo (or boundary) with respect to a process if at least one of its incident edges or vertices is assigned to a different process After initiating the receive operations, each process proceeds with blocking sends, transmitting the values of its local boundary vertices to the corresponding processes that require them for their own edge-based computations. This stage

ensures that all data dependencies related to the boundary edges are satisfied before local edge computations begin. The algorithm then waits for the completion of the non-blocking receives to guarantee the availability of remote data required for correct computation (line 5).

Following pre-communication, the local edge computation phase is performed, during which each process carries out computations over its locally assigned edges using both locally available and newly received halo vertex data.

The post-communication phase begins with local reduce operations, where each process partially aggregates contributions from its boundary edges that are associated with the same halo vertices. After computing these partial reductions, each process again posts non-blocking receives to collect reduced results from remote processes. Subsequently, it performs blocking sends to transmit its own partial results to the processes responsible for the corresponding halo vertices. The algorithm waits for the completion of these non-blocking receives to ensure the correctness of the final vertex-level aggregation (line 10).

Algorithm 1 Generic Edge-Vertex-Parallel Graph Computation

```
1: function GENERIC EDGE-VERTEX-PARALLEL
2:   while until some criteria met do
3:     IRECVs for values associated with my halo vertices
4:     SEND values associated with my boundary vertices
5:     WAITALL on IRECVs
6:     Perform edge computations associated with local edges
7:     Perform local reduce for each subset of edges having same halo vertices
8:     IRECVs for partially reduced results computed for my boundary vertices
9:     SEND results of local reduce operations
10:    WAITALL on IRECVs
11:    Perform aggregation of local/nonlocal edge computations results on vertices
12:    Perform computations associated with local vertices
13:  end while
14: end function
```

Once the post-communication completes, the aggregation phase is executed. Here, each process combines the results of local edge computations with the incoming partial aggregates to finalize the values associated with its local vertices. Finally, the algorithm proceeds to the vertex update phase, where computations

specific to each local vertex are carried out using the aggregated values.

From a local perspective in a GCN implementation, lines 3–11 of the algorithm correspond to the aggregation phase, while line 12 encapsulates the vertex-wise update phase under an edge-partitioned setting. From a global computation standpoint, lines 3–11 collectively represent a row-column-parallel SpMM, with an important constraint that the matrix entries a_{ij} and a_{ji} are co-located on the same process. Line 12 corresponds to a GeMM operation applied locally to update model parameters based on the aggregated features.

Chapter 5

Models and Methods

In this section, we describe how to obtain computationally balanced partitions with respect to both vertex and edge loads, utilizing the edge-vertex-based partitioning model for minimizing communication volume while balancing the computational load among processes. We describe and discuss the proposed bipartite graph model, a prior edge-clustering method, and finally, a vertex assignment method for communication load balancing.

5.1 Bipartite Graph Model

A given undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is modeled as a bipartite graph $\mathcal{B} = (\mathcal{U}_\mathcal{B} = \mathcal{U}_\mathcal{E} \cup \mathcal{U}_\mathcal{V},\ \mathcal{E}_\mathcal{B})$. Here, $\mathcal{U}_\mathcal{E}$ and $\mathcal{U}_\mathcal{V}$ denote the bipartition on the vertices of $\mathcal{B}$. To avoid confusion between the vertices of $\mathcal{G}$ and $\mathcal{B}$, we will henceforth refer to the vertices in $\mathcal{U}_\mathcal{B}$ as nodes. Moreover, we will distinguish between the nodes in $\mathcal{U}_\mathcal{E}$ and $\mathcal{U}_\mathcal{V}$ by referring to them as *edge-nodes* and *vertex-nodes*, respectively. B contains an *edge-node* u_{ij} for each edge (v_i, v_j) of $\mathcal{G}$, and a *vertex-node* u_i for each vertex v_i of $\mathcal{G}$. Note that we use a double index for an *edge-node* for the sake of clarity of discussion. In this way, in B, each *edge-node* u_{ij} connects the pair of *vertex-nodes* u_i and u_j. The adjacency relations for the *edge* and *vertex nodes* of

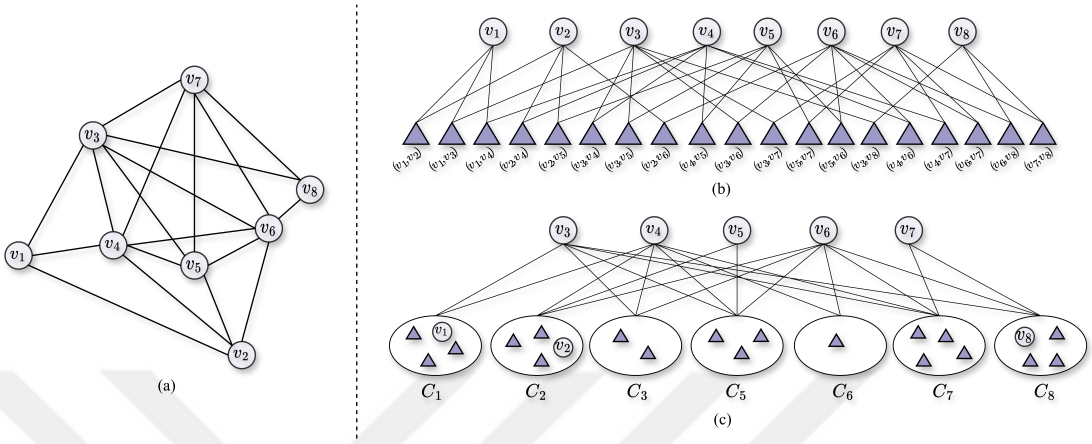


Figure 5.1: (a) An example graph with 8 vertices and 19 edges, (b) the proposed bipartite graph model with 27 nodes and 38 edges, (c) the bipartite graph resulting from the proposed edge-node clustering method now contains 12 nodes and 21 edges. Nodes v_1, v_2, and v_8 become internal vertices of clusters C_1, C_2, and C_8, respectively, while cluster C_4 is removed due to the absence of any associated edges.

B can be given as;

$$Adj_{\mathcal{B}}(u_i) = \{u_{ij} \in \mathcal{U}_{\mathcal{E}}\} = \{u_{ij} | (v_i, v_j) \in \mathcal{E}\} \quad (5.1)$$

$$Adj_{\mathcal{B}}(u_{ij}) = \{u_i, u_j \in \mathcal{U}_{\mathcal{V}}\} = \{v_i, v_j | (v_i, v_j) \in \mathcal{E}\} \quad (5.2)$$

As seen in Equations 5.1 and 5.2, each *vertex-node* u_i has $deg_{\mathcal{G}}(v_i)$ incident edges and each *edge-node* u_{ij} has two incident edges (i.e., (u_{ij}, u_i) and (u_{ij}, u_j)) in $\mathcal{B}$. Thus $\mathcal{B}$ contains $|\mathcal{V}|+|\mathcal{E}|$ nodes and $2\,|\mathcal{E}|$ edges. Without loss of generality, we assume that each edge has a unit cost.

We should note here that the proposed bipartite graph model is different than the bipartite graph model proposed by Hendrickson and Kolda, especially for rectangular matrices [37]. In their model, rows and columns of a sparse matrix constitute two different node sets of the bipartite graph.

The nice property of the proposed bipartite graph model is that it contains *edge-nodes* and *vertex-nodes* separately, thus enabling encoding balancing on the vertices and edges of $\mathcal{G}$. In the target parallel graph application, if there is no synchronization between computations associated with the graph vertices and edges, in $\mathcal{B}$, the *edge-nodes* and *vertex-nodes* are assigned different weights equal

to the amount of respective computations. That is

$$w(u_{ij}) = eload((v_i, v_j)) \tag{5.3}$$

$$w(u_i) = vload(v_i), \tag{5.4}$$

where $eload(\cdot)$ and $vload(\cdot)$ denote the amount of computation associated with graph edges and vertices, respectively. On the other hand, if there is an in-between synchronization, we can adopt a two-constraint formulation with setting node weights as $w_1(u_{ij}) = eload((v_i, v_j))$, $w_2(u_{ij}) = 0$ and $w_1(u_i) = 0$, $w_2(u_i) = vload(v_i)$.

Figure 5.1(a) illustrates a sample graph with 8 vertices and 19 edges. Figure 5.1(b) presents the proposed bipartite graph model corresponding to this sample graph. As shown, the bipartite model comprises 27 nodes, consisting of 8 *vertex-nodes* and 19 *edge-nodes*, and a total of 38 edges. As also seen in the figure, the degree of each *edge-node* equals 2 and the degree of each *vertex-node* equals the degree of the respective vertex of $\mathcal{G}$.

A given vertex partition $\Pi(\mathcal{B}) = \{\mathcal{U}^1, ..., \mathcal{U}^q, ..., \mathcal{U}^P\}$ is decoded as assigning all atomic tasks corresponding to the *edge-nodes* and the *vertex-nodes* in $\mathcal{U}^q$ to process P^q without loss of generality. For a given partition $\Pi(\mathcal{B})$, the weight of part $\mathcal{U}^q$ is defined as

$$W(\mathcal{U}^q) = \sum_{u_{ij} \in \mathcal{U}_{\mathcal{E}}{}^q} w(u_{ij}) + \sum_{u_i \in \mathcal{U}_{\mathcal{V}}{}^q} w(u_i), \tag{5.5}$$

which denotes the aggregate computational load of process P^q, assuming that part $\mathcal{U}^q$ is assigned to process P^q. The first and second weights of part $\mathcal{U}^q$ denote the computational load of P^q in the edge computation and the vertex computation phases, respectively.

Although the cutsize of a given partition $\Pi(\mathcal{B})$ of $\mathcal{B}$ relates to the total communication volume, the bipartite graph model fails to exactly encode the total communication volume.

5.2 Edge Clustering

The proposed bipartite graph model has a significantly larger size complexity, with $|\mathcal{V}|+|\mathcal{E}|$ nodes and $2|\mathcal{E}|$ edges, compared to the traditional 1D graph model, which contains only $|\mathcal{V}|$ vertices and $|\mathcal{E}|$ edges. Furthermore, because the number of edge-nodes significantly exceeds the number of *vertex-nodes*, the matching heuristics used during the coarsening phase of the multilevel partitioner tend to produce low-quality matching during the coarsening of a multilevel partitioner. In order to address these deficiencies of the bipartite graph model, we propose and describe a simple yet effective *edge-node* clustering method to be used before the partitioning.

We initialize $|\mathcal{U}_{\mathcal{V}}|$ empty edge-node clusters $\{C_1, C_2, ..., C_{|\mathcal{U}_{\mathcal{V}}|}\}$, one cluster per each *vertex-node*. That is, C_i is associated with the *vertex-node* u_i. In the proposed method, each *edge-node* is assigned to the cluster of the *vertex-node* with the smaller degree among its two incident *vertex-nodes*. That is, an edge u_{ij} is added to C_i if $deg(u_i) < deg(u_j)$ and it is added to C_j if $deg(u_j) < deg(u_i)$, where $deg(\cdot)$ denotes the degree of a *vertex-node*. Ties ($deg(u_i) = deg(u_j)$) are broken randomly.

During the *edge-node* clustering process, clusters associated with certain *vertex-nodes* may remain empty and are subsequently removed. Furthermore, if all *edge-nodes* incident to the same *vertex-node* are assigned to that *vertex-node's* cluster, then the *vertex-node* is considered internal to that cluster. The former and the latter cases usually happen for high-degree and low-degree *vertex-nodes* with low-degree and high-degree neighbors, respectively.

After the clustering, each non-empty cluster C_i is coalesced into a single supernode of the coarse bipartite graph to be provided to the graph partitioner. The adjacency of supernode s_i representing C_i is set to be equal to the union of the adjacencies of the constituent *edge-nodes* of s_i. The weight of supernode s_i is set to be equal to the sum of the weight of its constituent *edge-nodes* plus the

weight of *vertex-node* u_i if it becomes internal to C_i. That is

$$Adj_{\mathcal{B}*}(s_i) = \bigcup_{u_{ij} \in C_i} Adj_{\mathcal{B}}(u_{ij}) \tag{5.6}$$

$$w(s_i) = \sum_{u_{ij} \in C_i} w(u_{ij}) + w(u_i)\big|_{u_i \in C_i} \tag{5.7}$$

Figure 5.1(c) shows the result of the *edge-node* clustering on the sample bipartite graph given in Figure 5.1(b). Among the clustering of all e*dge-nodes*, tie-breaking occurred only in the clustering of u_{46}, u_{57}, and u_{36}. As seen in the figure, cluster C_4 disappears since all of the neighbor *edge-nodes* of *vertex-node* u_4 of degree 6 are assigned to other clusters. *Vertex-nodes* u_1, u_2, and u_8 become internal to C_1, C_2 and C_8 since all their neighbor *edge-nodes* are respectively assigned to C_1, C_2 and C_8.

The proposed bipartite graph partitioning model, combined with the *edge-node* clustering and a carefully designed edge weighting scheme, has the potential to mitigate the well-known deficiency of conventional graph partitioning models in accurately capturing true communication volume. We discuss this feature below.

During the conventional matching or clustering algorithms employed in the coarsening phase of multilevel partitioning tools, when two or more nodes are merged into a supernode, all multiple edges between any pair of such supernodes are collapsed into a single edge. The weight of this resulting edge is set equal to the number of original edges that previously connected nodes in the corresponding groups. Following this convention, whenever we have more than one *edge-node* connecting the same *vertex-node* v_i are assigned to C_i, the single edge connecting supernode s_i with v_i is assigned a weight equal to the count of such edges.

However, if we consider net coalescing after vertex clustering in the hypergraph model, which is known for accurately encoding true communication volume, multiple pins of a net that connect vertices within the same supervertex are coalesced into a single pin. Inspired by this desirable property of the hypergraph model, we propose assigning a unit weight to a coalesced edge that represents multiple original edges, thus better aligning edge weights with the actual communication volume.

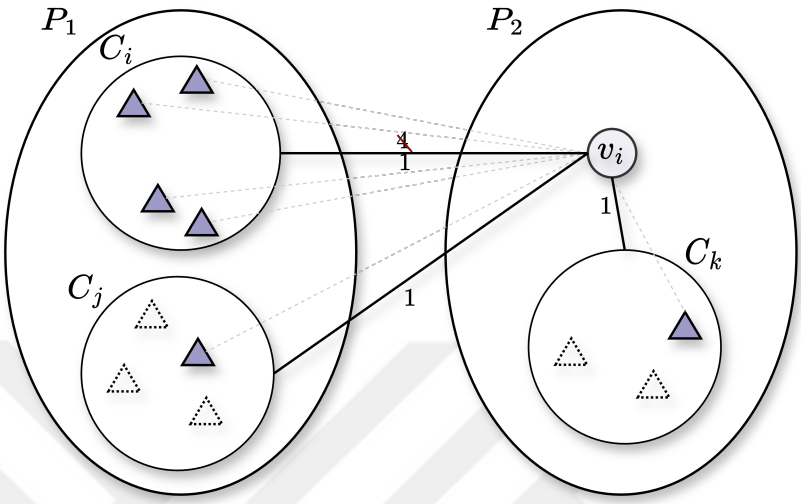


Figure 5.2: A 2-way partition of a given *vertex-node* and its neighbor *edge-node* clusters

In Figure 5.1(c), the weights of the coalesced edges are omitted for clarity, in order to avoid potential confusion in the illustration. In the figure, the coalesced edge between v_7 and C_7 has a weight of 4 and 1 in the conventional scheme and the proposed scheme, respectively.

We also provide Figure 5.2 to illustrate the proposed edge weighting scheme. In the figure, the edge clustering heuristic assigns four edges of v_i to cluster C_i, and the remaining two edges to clusters C_j and C_k. Under the given 2-way partition, an edge weight of 4 between v_i and C_i would incorrectly encode the communication volume across the cut edge (v_i, C_i) as $4+4$, while the proposed unit edge weighting correctly represents the communication volume as $1+1 = 2$. In the dissection of communication volume as $1 + 1$, due to cut edge (v_i, C_i), the first 1 refers to process P_2 sending the data associated with v_i to P_1 in the pre-communication phase, whereas the second 1 refers to P_1 performing local reduce on five neighbor edges clustered to C_i and C_j for the post-communication phase. The proposed model still overestimates the communication volume as $2 + 2$, due to the presence of two cut edges, (v_i, C_i) and (v_i, C_j). In reality, the correct communication volume is $1 + 1$, as P_2 needs to send v_i to P_1 only once and P_1 will send the result of the local reduce on four edges in C_i and one edge in C_j.

It is very clear that the proposed edge weighting scheme reduces the amount of overestimation from 10 to 4.

In the multilevel graph partitioning paradigm, all coarse graphs, including the initial flat graph, are typically refined during the uncoarsening phase. However, in the current implementation, although the coarse bipartite graph $\mathcal{B}_1$ at level 1, obtained by applying *edge-node* clustering to $\mathcal{B}_0$, is refined starting from the coarsest bipartite graph at the bottom of the V-cycle, the flat bipartite graph B_0 is not refined. In other words, B_0 lacks a corresponding uncoarsening level for refinement. To address this, we introduce an additional uncoarsening and refinement level. Specifically, we uncoarsen the partition $\Pi(\mathcal{B}_1)$ of $\mathcal{B}_1$ to obtain an initial partition $\Pi(\mathcal{B}_0)$ on $\mathcal{B}_0$, and then perform a P-way refinement on $\Pi(\mathcal{B}_0)$, using an approach similar to the k-way refinement heuristic described in Metis [38].

5.3 Communication Volume Balancing

As mentioned earlier, vertex balancing loosely relates to communication volume balance, since boundary vertices are the main reason for the communication volume imbalance, because of their high degrees. Consider a vertex owned by a process $\mathcal{U}^q/P^q$. v_i will incur a computational load of $vload(v_i)$ to P^q, whereas it will incur a send volume load of $size(v_i)(\lambda(v_i)-1)$ to P^q only if it is a boundary vertex of process $\mathcal{U}^q$. Here, size$(\cdot)$ denotes the amount of data associated with a vertex, and $\lambda(\cdot)$ represents its connectivity in the given partition—that is, the number of distinct processes it connects to. Specifically, $\lambda(v_i)-1$ refers to the number of distinct processes other than P^q in which incident edges of vertex v_i reside.

To address the reassignment of boundary vertices, we adopt a bin-packing heuristic akin to the best-fit decreasing strategy, which is widely used for approximating solutions to the *NP*-hard *P*-feasible bin packing problem [39]. The main steps of the proposed vertex-to-process assignment heuristic are as follows: Given a partition $\Pi(\mathcal{G})$, we visit the boundary vertices in increasing order with

respect to their connectivities. Here, the connectivity of a boundary vertex is interpreted as its flexibility in process assignment. At each step, we consider reassigning vertex v_i, in the given order, to a feasible bin/process in $\Lambda(v_i)$ that has the least send volume load, which serves as the best-fit criterion. Here, feasibility refers to a process that does not disturb the computational load constraint after a possible reassignment. $\Lambda(v_i)$ denotes the set of distinct processes connected by v_i, where $|\Lambda(v_i)| = \lambda(v_i)$. Restricting the reassignment of v_i to processes in $\Lambda(v_i)$ is intended to avoid increasing the total communication volume achieved by the initial partition $\Pi(\mathcal{G})$.

Chapter 6

Dataset and Experimental Setup

6.1 Dataset

We perform our experiments on three datasets.

First, we collected more than 600 graphs/matrices from the Suite Sparse Matrix Collection [40] with a number of vertices/rows between $5,000$ and $10,000,000$ to create a wide range of data pool. The first dataset, referred to as `ds-highcv`, comprises 175 graphs with a *coefficient of variation (cv)* greater than 1. We categorized these graphs into 6 groups according to their increasing *cv* values. The second dataset, referred to as `ds-highsp`, includes 212 graphs with a greater sparsity value (maximum degree over number of vertices) 0.01. Similar to the first one, we categorized these graphs into 5 groups according to their increasing *sparsity* values.

The naming for the categories is in the format of `G-Mm-Vv`. Here, $\texttt{m} \in \{\texttt{CV}, \texttt{SP}\}$ denotes the classification metric of the dataset and `v` denotes the corresponding value of this metric. For example, `G-CV-200` includes graphs that have *cv* value of at least 2. Note that `G-Mm-2000` $\subseteq$ `G-Mm-1000` $\subseteq$ ... $\subseteq$ `G-Mm-100` and `G-Mm-100`

Table 6.1: Properties of the `ds-highcv` and `ds-highsp` datasets.

Graph category	Number of graphs	Avg number of vertices	Avg number of edges	Avg max degree	Avg mean degree	Avg max deg. ratio
G-CV-100	175	79 524	1 202 352	14 194	29	0.203
G-CV-150	146	78 836	1 107 804	16 840	25	0.230
G-CV-200	121	76 485	1 017 947	20 089	23	0.266
G-CV-500	62	113 691	1 093 348	35 949	16	0.371
G-CV-1000	31	140 562	1 060 110	63 572	10	0.458
G-CV-2000	13	248 689	1 694 520	130 600	9	0.566
G-SP-001	212	60 849	1 460 518	12 321	46	0.177
G-SP-005	136	58 956	1 193 451	8 112	32	0.121
G-SP-010	90	59 230	1 147 336	7 196	31	0.101
G-SP-020	62	61 414	1 381 689	6 351	26	0.094
G-SP-050	10	162 534	2 178 496	4 716	23	0.076

contains all the matrices in the corresponding dataset. The top section of Table 6.1 displays various important properties of the graphs in those categories as the geometric averages. Our motivation for this selection and categorization is to evaluate the performance of our proposed models using graphs that are highly scale-free and irregularly sparse, which are particularly difficult to partition using conventional 1D vertex partitioning.

In the third dataset, namely `ds-general`, we used 22 publicly available graphs for the evaluation of parallel 2D GCN training, with characteristics listed in Table 6.2. Eight of these are datasets from the Open Graph Benchmark, available in PyG Datasets, and commonly used for GNN tasks [41, 23]. Eight of these are datasets from the Open Graph Benchmark, available in PyG Datasets, and commonly used for GNN tasks [41, 23]. These datasets are briefly described below:

- `ogbn-products`: This dataset is an undirected, unweighted graph modeling Amazon's product co-purchasing network [41]. Each node corresponds to a product sold on Amazon, and an edge between two nodes signifies that the products are frequently bought together. Node features are derived

from bag-of-words representations of product descriptions, which are subsequently reduced to 100 dimensions using Principal Component Analysis (PCA).

- `ogbn-arxiv`: The `ogbn-arxiv` represents the citation network among all Computer Science (CS) papers on arXiv indexed by Microsoft Academic Graph (MAG) [42]. Nodes correspond to individual papers, and edges denote citation relationships between them. Each paper is associated with a 128-dimensional feature vector, obtained by averaging the word embeddings of its title and abstract. These word embeddings are generated using the skip-gram model [43] trained on the MAG corpus.

- `ogbn-mag`: This is a heterogeneous graph derived from a subset of the MAG [42]. It includes four types of nodes—papers, authors, institutions, and fields of study and four types of directed relationships: authors are affiliated with institutions, authors write papers, papers cite other papers, and papers have a topic of a field of study. Similar to `ogbn-arxiv`, each paper node is assigned a 128-dimensional feature vector, while the remaining entity types do not have associated input features. We used this dataset as it is a homogeneous one, simply by avoiding the types of nodes.

- `Yelp`: This dataset is an undirected, unweighted graph constructed from the Yelp Challenge dataset [44]. Each node represents a business listed on Yelp, and edges indicate that two businesses are frequently reviewed by the same user. Node features are derived from bag-of-words representations of business attributes, descriptions, and categories, reduced to 300 dimensions.

- `Amazon`: This dataset models Amazon's product co-review network [45]. Each node corresponds to a product on Amazon, and an edge between two nodes denotes that the products have been reviewed by the same user. Node features are generated from product metadata (e.g., title, category, and description) encoded as bag-of-words vectors and compressed to 200 dimensions.

- `Reddit`: The `Reddit` dataset captures the post-to-post interaction graph on Reddit [46]. Each node represents a post made on Reddit, and an edge

Table 6.2: Properties of the graphs in the `ds-general` dataset.

	Number of		Row/column degree		
Matrix	rows/cols	nonzeros	max	cv	maxdr
`Amazon`	1 569 960	264 339 468	75 134	4.983	0.048
`Reddit`	232 965	114 848 857	21 658	1.622	0.093
`Reddit2`	232 965	23 213 838	3 649	1.258	0.016
`ogbn-arxiv`	169 343	2 315 598	13 161	5.020	0.078
`ogbn-mag`	1 134 649	42 180 539	736 389	20.825	0.649
`ogbn-products`	2 449 029	126 167 053	17 482	1.862	0.007
`Yelp`	716 847	13 954 819	4 886	3.641	0.007
`as-Skitter`	1 696 415	22 190 596	35 455	10.463	0.021
`caiduRouterLevel`	192 244	1 218 132	1 071	2.232	0.006
`citationCiteseer`	268 495	2 313 294	1 318	1.890	0.005
`cit-Patents`	3 774 768	16 518 948	770	1.778	0.000
`coAuthorsCiteseer`	227 320	1 628 268	1 372	1.485	0.006
`coAuthorsDBLP`	299 067	1 955 352	336	1.501	0.001
`com-Amazon`	334 863	1 851 744	7 061	1.042	0.002
`com-DBLP`	317 080	2 099 732	343	1.511	0.001
`com-LiveJournal`	3 997 962	69 362 378	14 815	2.476	0.004
`com-Orkut`	3 072 441	234 370 166	33 313	2.029	0.011
`com-Youtube`	1 134 890	5 975 248	28 754	9.640	0.025
`coPapersDBLP`	540 486	30 491 458	3 299	1.174	0.006
`hollywood-2009`	1 139 905	113 891 327	11 468	2.721	0.010
`kmer_V2a`	55 042 369	117 217 600	39	0.316	0.000
`wikipedia-20070206`	3 566 907	45 030 389	7 061	2.612	0.002

cv: coefficient of variation. **maxdr**: maximum degree ratio (i.e., max degree divided by the number of rows/columns).

is created between two posts if the same user comments on both. Nodes are annotated with 602-dimensional features derived from the average word embeddings of the post's title and body text, capturing semantic information from user-generated content.

The others are sparse and scale-free matrices from the Suite Sparse Matrix Collection, representing various graph types such as social media, citation networks, and genomics [40].

For some datasets, features and labels were not provided, so, as in other studies, we generated k values between 32 and 64 and set F to 100 [47, 48].

6.2 Implementation and System Details

We implemented a GNN full-batch training framework using the C programming language coupled with MPI for interprocess communication, and it supports both one-phase and two-phase communication schemes. We utilized Combinatorial BLAS [49] to obtain superior performance in linear algebraic operations. We use two Tier-0 systems in our experiments. The first system, LUMI, is an HPE Cray EX supercomputer with an underlying Dragonfly topology. Each compute node is equipped with two AMD EPYC 7763 CPUs with 64 cores each running at 2.45 GHz for a total of 128 cores per node. All LUMI compute nodes use the HPE Cray Slingshot-11 200 Gbps network interconnect. The second system, MareNostrum5, comprises 6,408 nodes; each node is powered by dual Intel Sapphire Rapids 8480+ processors running at 2 GHz with 56 cores per processor (totaling 112 cores per node), paired with 256 GB of DDR5 memory. The network topology used is fat-tree, with islands with full fat tree without contention of 2160 nodes, and with a contention between islands of 2/3.

In our experiments, we used Metis for partitioning the conventional graph model as well as the proposed models in default settings. To further enhance the efficiency of the partitioning step, we consider parallel graph partitioners such as ParMETIS [6] and PT-Scotch [2]. These tools are designed to significantly speed up the partitioning process, which is crucial as an effective implementation must quickly amortize the time spent on partitioning to accelerate the overall training phase. By optimizing the partitioning step, we aim to significantly improve the efficiency and speed of the training process, ensuring that the computational resources are utilized most effectively.

6.3 Evaluation Metrics

To assess the effectiveness of different graph partitioning strategies, we evaluate them using a set of well-established metrics that capture both communication

overhead and computational load balance. These metrics are critical for understanding the scalability and performance characteristics of parallel graph processing frameworks. In particular, we categorize the metrics into two main groups: *communication metrics*, which quantify the cost and distribution of data exchange between processing units, and *computational metrics*, which measure how evenly the computational workload is distributed across parts/processes. Together, these metrics provide a comprehensive view of the trade-offs involved in partitioning and help identify strategies that best support efficient and scalable execution on distributed systems.

6.3.1 Communication Volume Load

Average Volume

The *Average Communication Volume* (ACV) metric quantifies the expected communication overhead incurred by a partitioning scheme in a distributed setting. For each vertex v, we define its communication volume as $|\text{parts}(v)| - 1$, where $\text{parts}(v)$ denotes the set of parts in which v is replicated. This value reflects the number of processes that need to v during computation. The average is then taken over the sum of communication volume incurred by each vertex divided by the number of processes to obtain the ACV. This metric serves as a normalized indicator of overall communication cost: lower ACV values imply better locality and reduced inter-process communication, which are critical for the scalability of parallel graph computations.

Maximum Volume

The *Maximum Communication Volume* metric captures the worst-case communication load among all processes. For each vertex v, the communication volume is defined as $|\text{parts}(v)| - 1$, representing the number of parts that require access to v. The maximum volume is the largest such value observed across all

processes. This metric highlights potential communication bottlenecks by identifying the most highly replicated vertices, which can lead to excessive synchronization and message-passing overhead. In distributed environments, minimizing the maximum volume is important for avoiding stragglers and achieving balanced communication performance.

Communication Load Imbalance

Communication Imbalance measures the unevenness in communication load across all processes. It is defined as the ratio between the maximum communication volume handled by any process and the average communication volume across all processes. Let c_i denote the communication volume for process i, then the imbalance is computed as

$$\frac{\max_i c_i}{\frac{1}{P}\sum_{i=1}^{P} c_i},$$

where P is the total number of processes. While this metric is generally useful for quantifying load imbalance, it can be misleading when comparing different partitioning strategies that result in significantly different total or average communication volumes. In such cases, a lower imbalance ratio may merely reflect a lower denominator (average load), rather than genuinely better load distribution.

However, we used this metric only to assess the performance of our communication balancing heuristic, so that in our case, this issue is mitigated. The communication balancing heuristic is used in conjunction with an edge partitioning derived from our edge-clustering method. This heuristic improves load balance by reducing the maximum send and receive volumes without altering the total or average communication volume. As a result, the denominator in the imbalance ratio remains constant, and any improvement in the metric directly reflects a more uniform distribution of communication load. Under this constraint, the communication imbalance metric becomes a fair and meaningful indicator of load distribution effectiveness.

6.3.2 Computational Load Imbalance

Computational Load Balance (also referred to as computational imbalance) assesses how evenly the computational workload is distributed across processes. It is typically measured as the ratio between the maximum workload assigned to any partition and the average workload across all processes. If w_i denotes the computational load of part i, then the imbalance is defined as

$$\frac{\max W_p}{\frac{1}{P}\sum_{i=1}^{P} W_i}, \tag{6.1}$$

where P is the number of parts. The computational load is usually estimated using vertex weights, edge counts, or application-specific cost models. An imbalance value close to 1 indicates near-perfect load balance, whereas higher values reflect skewed distributions that can lead to idle processes and degraded parallel performance. This metric is essential for evaluating the efficiency and scalability of partitioning strategies, especially in compute-intensive graph applications.

Chapter 7

Results and Comparison

7.1 Compared schemes and models

We evaluate our proposed models and methods in three schemes. These schemes are as follows:

- `BG`: The bipartite graph (BG) model proposed in this work. It represents the initial formulation without any additional enhancements.
- `BG-EC`: The bipartite graph model augmented with our proposed a priori edge clustering (EC) method, which groups edges before partitioning to improve structural locality in the coarsening phase.
- `BG-EC-CB`: The `BG-EC` model is further extended with our communication balancing (CB) heuristic, which aims to reduce communication volume imbalance across processes.

The baseline models that we compare our proposed models against are as follows:

- `1D`: A traditional one-dimensional (1D) vertex-partitioning model, which distributes vertices across partitions and assigns edges accordingly. This

model serves as a classical baseline widely used in distributed graph processing.

- `KahipNE`: A recent vertex-partitioning model based on the Node-Edge balancing heuristic, implemented using the KaHIP framework. It aims to improve edge balance and locality while controlling vertex replication.

In Section 7.2, we compare the three proposed schemes among themselves and against the conventional model `1D`. We further evaluate the proposed schemes in comparison with the `KahipNE` *node-edge balanced* vertex-partitioning scheme. In Section 7.3, we analyze how each scheme scales under increasing processor counts in a distributed-memory setting by running a parallel GCN training application on multiple clusters. We evaluate their scalability and relative performance against each other across various processor configurations and system scales.

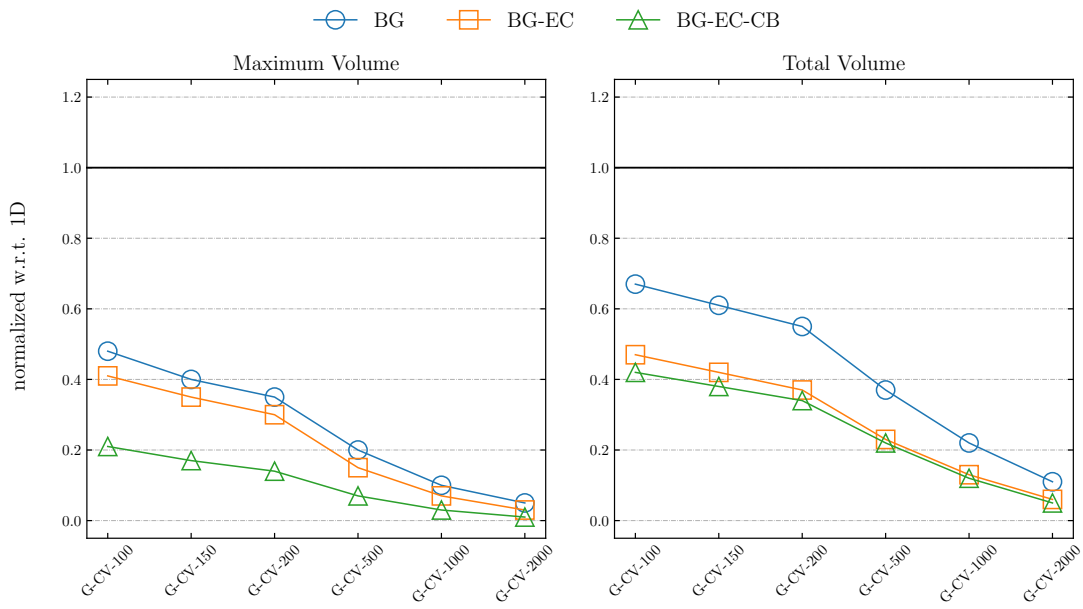
7.2 Partitioning Quality Analysis

In this section, we provide the results of the proposed partitioning schemes in terms of maximum send/recv volumes, total (average) volume, maximum send plus recv (S+R) volume, and the computational imbalance ratios on datasets `ds-highcv`, `ds-highsp`, `ds-general` following the categorization described in Section 6.1. The results obtained by `BG`, `BG-EC`, and `BG-EC-CB` are normalized with respect to those of `1D`, the obtained normalized values for $P = 256$ are displayed as plots for both of the metrics separately in Figure 7.1, where Figure 7.1(a) displays the results obtained by the `ds-highcv` and Figure 7.1(b) displays the results obtained by the `ds-highsp`. The detailed results for each $P \in \{64, 128, 256, 512, 1024\}$ are given in Tables 7.1 and 7.2. In the plots in Figure 7.1, the x-axis denotes the respective matrix categories in order of increasing `CV` or `SP`, whereas the y-axis denotes the normalized values. Each value in the plots and the tables is the geometric average of the normalized values obtained for the matrices in the respective category. For example, the three values reported for category `G-CV-500` in the plot for maximum volume (the left one in

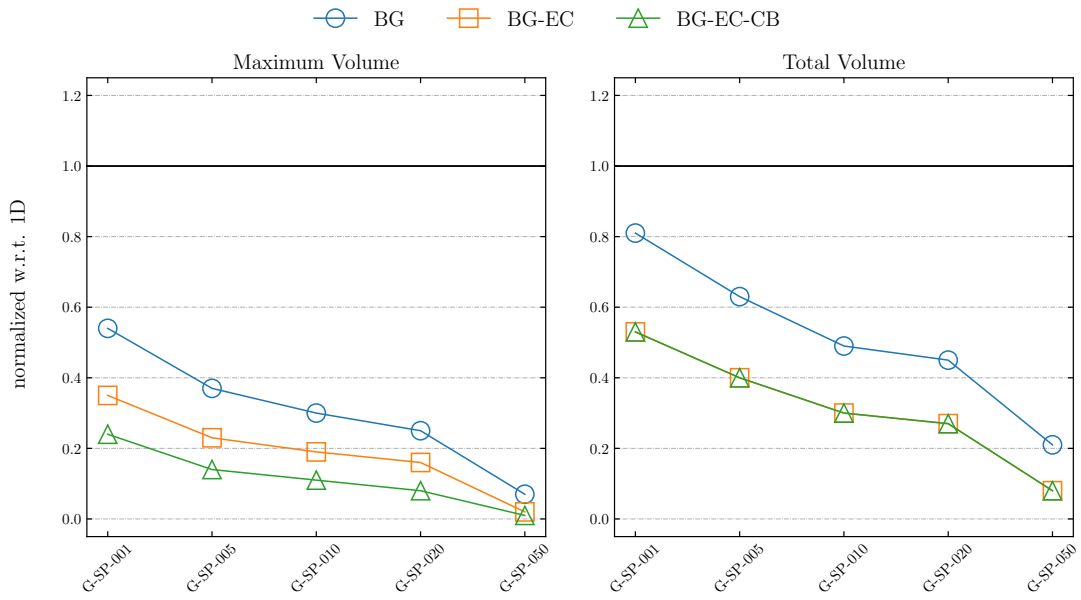
Figure 7.1(a)) denote the geometric averages of the normalized values obtained by BG, BG-EC, and BG-EC-CB with respect to those obtained by 1D on 121 matrices in this category. The actual and normalized results obtained from ds-general for each $P \in \{64, 128, 256, 512, 1024, 2048, 4096\}$ are given in Table 7.3 in detail. In Figure 7.2, the results obtained by BG, BG-EC, and BG-EC-CB are compared with the result of the KahipNE scheme in terms of maximum volume, total volume, and computational imbalance on ds-highcv for each $P \in \{64, 128, 256, 512, 1024\}$. In the bar graphs in Figure 7.2, the x-axis denotes the respective matrix categories in order of increasing CV, whereas the y-axis denotes the actual values. Each row in Figure 7.2 corresponds to a P value.

As seen in Figure 7.1, BG, BG-EC, and BG-EC-CB perform drastically better than 1D in terms of both maximum and total volumes. The improvements obtained by all three schemes increase with increasing significance of the coefficient of variations and sparsity. For example, for category G-SP-001, BG, BG-EC, and BG-EC-CB respectively achieve improvements of 46%, 65% and 76%, whereas for category G-SP-050, these improvements increase to respectively, 93%, 98% and 99%. In addition, for category G-CV-Vv with varying v values of $100, 150, 200, 500, 1000$ and 2000, the improvement of BG over 1D gradually increases as 52%, 60%, 65%, 80%, 90% and 95%, respectively. The reason for this increase in the improvement is the insufficient performance of 1D models on the task of partitioning highly scale-free graphs, which results in both high maximum and total volumes.

When we compare BG, BG-EC, and BG-EC-CB among themselves, BG-EC-CB typically yields the most significant improvements in terms of maximum communication volume. This is primarily due to the incorporation of the communication balancing heuristic, which is specifically designed to minimize the peak communication load by distributing the communication volume more evenly across partitions. In terms of total communication volume, both BG-EC and BG-EC-CB achieve significantly lower volume values compared to BG, while the difference between BG-EC and BG-EC-CB is either negligible or very minor. As shown in Figure 7.1 (b), moving from G-SP-001 to G-SP-050, the improvement of BG increases from 19% to 79%, whereas the improvements of BG-EC and BG-EC-CB



(a)



(b)

Figure 7.1: Maximum volume and total volume of the proposed graph schemes `BG`, `BG-EC` and `BG-EC-CB` normalized with respect to those of `1D` on (a) `ds-highcv` and (b) `ds-highsp` for $P = 256$, averaged on matrices in each category.

increase from 47% to 92%. This difference can be attributed to the proposed edge-clustering method, which enables the multilevel partitioner to perform vertex matching in the early coarsening stages in a more informed manner. As a result, the partitioner is guided toward a solution space where a better cutsize can be achieved. The reason why `BG-EC` and `BG-EC-CB` exhibit similar total volume values is that the applied communication balancing heuristic typically does not affect the total volume. Even when it does, its effect is positive—for instance, by mitigating cases where the partitioner assigns certain *vertex-nodes* to different parts than their incident *edge-nodes* in order to achieve better computational balance.

As seen in Tables 7.1 and 7.2, the improvements of the proposed models in computational balance increase as the number of processes increases. For example, for category `G-CV-2000` with increasing number of processes $64, 128, 256, 512$ and 1024, the improvements of `BG` over `1D` respectively increase as 13%, 41%, 68%, 84% and 92%. The reason for the better performance of the proposed models at larger process counts is that the `1D` scheme suffers from limited granularity. As the number of parts increases, particularly in the case of scale-free graphs, the `1D` vertex-partitioning approach struggles to maintain computational balance due to the presence of high-degree vertices and its coarse-grained view of the computation structure. In contrast, the proposed bipartite-graph-based models offer finer-grained control over task (*edge-node*) and data (*vertex-node*) assignment, enabling more balanced workload distribution across processes as the system scales.

As shown in Table 7.3, the proposed models achieve the improvement in maximum communication volume primarily by decreasing the receive volume. For instance, with 4096 processes, the `BG-EC` scheme reduces the maximum send volume by 25%, while it reduces the maximum receive volume by 51%. This is because very high-degree vertices tend to have a large number of neighboring vertices that reside in different partitions, which contributes more heavily to the receive load than to the send load. However, this heavy receive-load problem is effectively alleviated by the fine-grained structure of the bipartite-graph-based models.

Table 7.1: Normalized values of maximum volume, total volume, and the computational imbalances of the proposed schemes `BG`, `BG-EC`, and `BG-EC-CB` with respect to those of `1D` on `ds-highcv` for $P \in \{64, 128, 256, 512, 1024\}$.

		Normalized values w.r.t. those of 1D								
		Maximum Volume			Total Volume			Comp. Imbalance		
P	Category	BG	BG-EC	BG-EC-CB	BG	BG-EC	BG-EC-CB	BG	BG-EC	BG-EC-CB
64	G-CV-100	0.47	0.29	0.23	0.58	0.33	0.33	0.96	0.98	0.99
	G-CV-150	0.40	0.24	0.18	0.51	0.29	0.29	0.96	0.98	0.98
	G-CV-200	0.34	0.20	0.15	0.44	0.24	0.24	0.96	0.98	0.98
	G-CV-500	0.19	0.10	0.07	0.25	0.13	0.13	0.94	0.96	0.96
	G-CV-1000	0.10	0.04	0.03	0.15	0.07	0.07	0.91	0.93	0.93
	G-CV-2000	0.06	0.02	0.01	0.08	0.03	0.03	0.87	0.89	0.89
128	G-CV-100	0.46	0.31	0.21	0.63	0.37	0.37	0.90	0.93	0.93
	G-CV-150	0.39	0.26	0.16	0.57	0.33	0.33	0.89	0.91	0.92
	G-CV-200	0.33	0.22	0.13	0.50	0.28	0.28	0.87	0.90	0.90
	G-CV-500	0.18	0.11	0.07	0.31	0.16	0.16	0.79	0.81	0.81
	G-CV-1000	0.09	0.05	0.03	0.18	0.09	0.09	0.69	0.71	0.71
	G-CV-2000	0.05	0.02	0.01	0.10	0.04	0.04	0.59	0.60	0.60
256	G-CV-100	0.48	0.34	0.21	0.67	0.42	0.42	0.74	0.76	0.77
	G-CV-150	0.40	0.29	0.17	0.61	0.38	0.38	0.71	0.72	0.73
	G-CV-200	0.35	0.26	0.14	0.55	0.34	0.34	0.66	0.68	0.68
	G-CV-500	0.20	0.14	0.07	0.37	0.22	0.22	0.51	0.53	0.53
	G-CV-1000	0.10	0.06	0.03	0.22	0.12	0.12	0.41	0.42	0.42
	G-CV-2000	0.05	0.02	0.01	0.11	0.05	0.05	0.32	0.33	0.33
512	G-CV-100	0.49	0.40	0.24	0.74	0.48	0.48	0.56	0.58	0.58
	G-CV-150	0.42	0.35	0.20	0.69	0.45	0.45	0.51	0.52	0.52
	G-CV-200	0.36	0.31	0.17	0.64	0.41	0.41	0.45	0.46	0.46
	G-CV-500	0.19	0.19	0.09	0.45	0.30	0.30	0.30	0.30	0.30
	G-CV-1000	0.09	0.09	0.04	0.29	0.18	0.18	0.22	0.22	0.22
	G-CV-2000	0.04	0.03	0.02	0.15	0.08	0.08	0.16	0.16	0.16
1024	G-CV-100	0.56	0.46	0.30	0.83	0.56	0.56	0.36	0.38	0.38
	G-CV-150	0.49	0.41	0.26	0.80	0.54	0.54	0.31	0.32	0.32
	G-CV-200	0.42	0.38	0.23	0.74	0.51	0.51	0.26	0.27	0.27
	G-CV-500	0.24	0.24	0.13	0.54	0.38	0.38	0.16	0.16	0.16
	G-CV-1000	0.12	0.12	0.07	0.37	0.24	0.24	0.11	0.11	0.11
	G-CV-2000	0.05	0.05	0.02	0.19	0.11	0.11	0.08	0.08	0.08

Table 7.2: Normalized values of maximum volume, total volume, and the computational imbalances of the proposed schemes `BG`, `BG-EC`, and `BG-EC-CB` with respect to those of `1D` on `ds-highsp` for $P \in \{64, 128, 256, 512, 1024\}$.

		Normalized values w.r.t. those of 1D								
		Maximum Volume			Total Volume			Comp. Imbalance		
P	Category	BG	BG-EC	BG-EC-CB	BG	BG-EC	BG-EC-CB	BG	BG-EC	BG-EC-CB
64	G-SP-001	0.58	0.33	0.27	0.72	0.45	0.45	0.96	0.99	0.99
	G-SP-005	0.38	0.20	0.15	0.51	0.29	0.29	0.96	0.98	0.98
	G-SP-010	0.28	0.14	0.09	0.37	0.19	0.19	0.95	0.97	0.97
	G-SP-020	0.23	0.10	0.07	0.31	0.14	0.14	0.94	0.96	0.96
	G-SP-050	0.09	0.02	0.01	0.18	0.04	0.04	0.78	0.80	0.80
128	G-SP-001	0.55	0.33	0.24	0.77	0.49	0.49	0.91	0.94	0.94
	G-SP-005	0.37	0.20	0.14	0.58	0.34	0.34	0.88	0.91	0.91
	G-SP-010	0.28	0.15	0.09	0.44	0.24	0.24	0.84	0.87	0.87
	G-SP-020	0.22	0.12	0.06	0.38	0.19	0.19	0.80	0.81	0.82
	G-SP-050	0.08	0.02	0.01	0.19	0.05	0.05	0.43	0.44	0.44
256	G-SP-001	0.54	0.35	0.24	0.81	0.53	0.53	0.78	0.80	0.80
	G-SP-005	0.37	0.23	0.14	0.63	0.40	0.40	0.69	0.71	0.71
	G-SP-010	0.30	0.19	0.11	0.49	0.30	0.30	0.58	0.59	0.60
	G-SP-020	0.25	0.16	0.08	0.45	0.27	0.27	0.47	0.48	0.48
	G-SP-050	0.07	0.02	0.01	0.21	0.08	0.08	0.23	0.24	0.24
512	G-SP-001	0.56	0.42	0.28	0.87	0.60	0.60	0.62	0.63	0.64
	G-SP-005	0.38	0.30	0.18	0.72	0.48	0.48	0.48	0.49	0.49
	G-SP-010	0.31	0.28	0.14	0.58	0.39	0.39	0.34	0.35	0.35
	G-SP-020	0.26	0.26	0.11	0.53	0.36	0.36	0.26	0.27	0.27
	G-SP-050	0.07	0.04	0.02	0.25	0.12	0.12	0.12	0.13	0.13
1024	G-SP-001	0.61	0.49	0.34	0.95	0.67	0.67	0.43	0.44	0.44
	G-SP-005	0.43	0.38	0.25	0.83	0.60	0.60	0.27	0.28	0.28
	G-SP-010	0.38	0.39	0.21	0.69	0.51	0.51	0.18	0.18	0.18
	G-SP-020	0.33	0.37	0.18	0.64	0.48	0.48	0.13	0.14	0.14
	G-SP-050	0.07	0.07	0.04	0.31	0.18	0.18	0.06	0.07	0.07

Table 7.3: Actual and normalized values of maximum send/receive volumes, average volume, and maximum send plus receive (S+R) volume of the proposed schemes `BG`, `BG-EC`, and `BG-EC-CB` with respect to those of `1D` on `ds-general` for $P \in \{64, 128, 256, 512, 1024, 2048, 4096\}$.

		actual values					**normalized w.r.t. 1D**				
		Max vol load per proc			Vol.	time	Max load per proc			Vol.	time
P	*Scheme*	Send	Recv	S+R	avg	(ms)	Send	Recv	S+R	avg	(ms)
64	`1D`	60 712	71 763	35 869	128 517	0.1013	-	-	-	-	-
	`BG`	62 938	62 938	35 491	125 876	0.0874	1.04	0.88	0.99	0.98	0.86
	`BG-EC`	40 123	40 123	21 626	80 246	0.0814	0.66	0.56	0.60	0.62	**0.80**
	`BG-EC-CB`	35 685	35 685	21 621	71 371	0.0838	0.59	0.50	0.60	0.56	0.83
128	`1D`	45 470	56 878	22 898	98 088	0.0622	-	-	-	-	-
	`BG`	47 172	47 172	23 036	94 345	0.0525	1.04	0.83	1.01	0.96	0.84
	`BG-EC`	30 047	30 047	13 482	60 094	0.0474	0.66	0.53	0.59	0.61	**0.76**
	`BG-EC-CB`	24 258	24 258	13 480	48 516	0.0480	0.53	0.43	0.59	0.49	0.77
256	`1D`	31 505	42 494	14 294	70 073	0.0385	-	-	-	-	-
	`BG`	34 823	34 823	14 520	69 645	0.0325	1.11	0.82	1.02	0.99	0.85
	`BG-EC`	20 990	20 990	8 296	41 981	0.0270	0.67	0.49	0.58	0.60	0.70
	`BG-EC-CB`	16 158	16 158	8 294	32 317	0.0269	0.51	0.38	0.58	0.46	**0.70**
512	`1D`	23 192	31 060	8 729	50 266	0.0321	-	-	-	-	-
	`BG`	24 359	24 359	8 908	48 719	0.0267	1.05	0.78	1.02	0.97	0.83
	`BG-EC`	15 327	15 327	5 049	30 655	0.0230	0.66	0.49	0.58	0.61	0.72
	`BG-EC-CB`	10 675	10 675	5 048	21 350	0.0228	0.46	0.34	0.58	0.42	**0.71**
1024	`1D`	16 547	23 414	5 234	36 831	0.0233	-	-	-	-	-
	`BG`	18 278	18 278	5 412	36 556	0.0196	1.10	0.78	1.03	0.99	0.84
	`BG-EC`	11 166	11 166	3 011	22 331	0.0146	0.67	0.48	0.58	0.61	0.63
	`BG-EC-CB`	7 205	7 205	3 004	14 409	0.0138	0.44	0.31	0.57	0.39	**0.59**
2048	`1D`	9 845	13 861	2 583	21 182	0.0137	-	-	-	-	-
	`BG`	12 031	12 031	2 719	24 063	0.0131	1.22	0.87	1.05	1.14	0.95
	`BG-EC`	6 595	6 595	1 477	13 190	0.0089	0.67	0.48	0.57	0.62	0.65
	`BG-EC-CB`	4 137	4 137	1 504	8 274	0.0083	0.42	0.30	0.58	0.39	**0.60**
4096	`1D`	6 630	10 238	1 617	15 032	0.0098	-	-	-	-	-
	`BG`	9 352	9 352	1 745	18 703	0.0104	1.41	0.91	1.08	1.24	1.06
	`BG-EC`	4 978	4 978	918	9 956	0.0067	0.75	0.49	0.57	0.66	0.69
	`BG-EC-CB`	3 049	3 049	942	6 099	0.0058	0.46	0.30	0.58	0.41	**0.59**

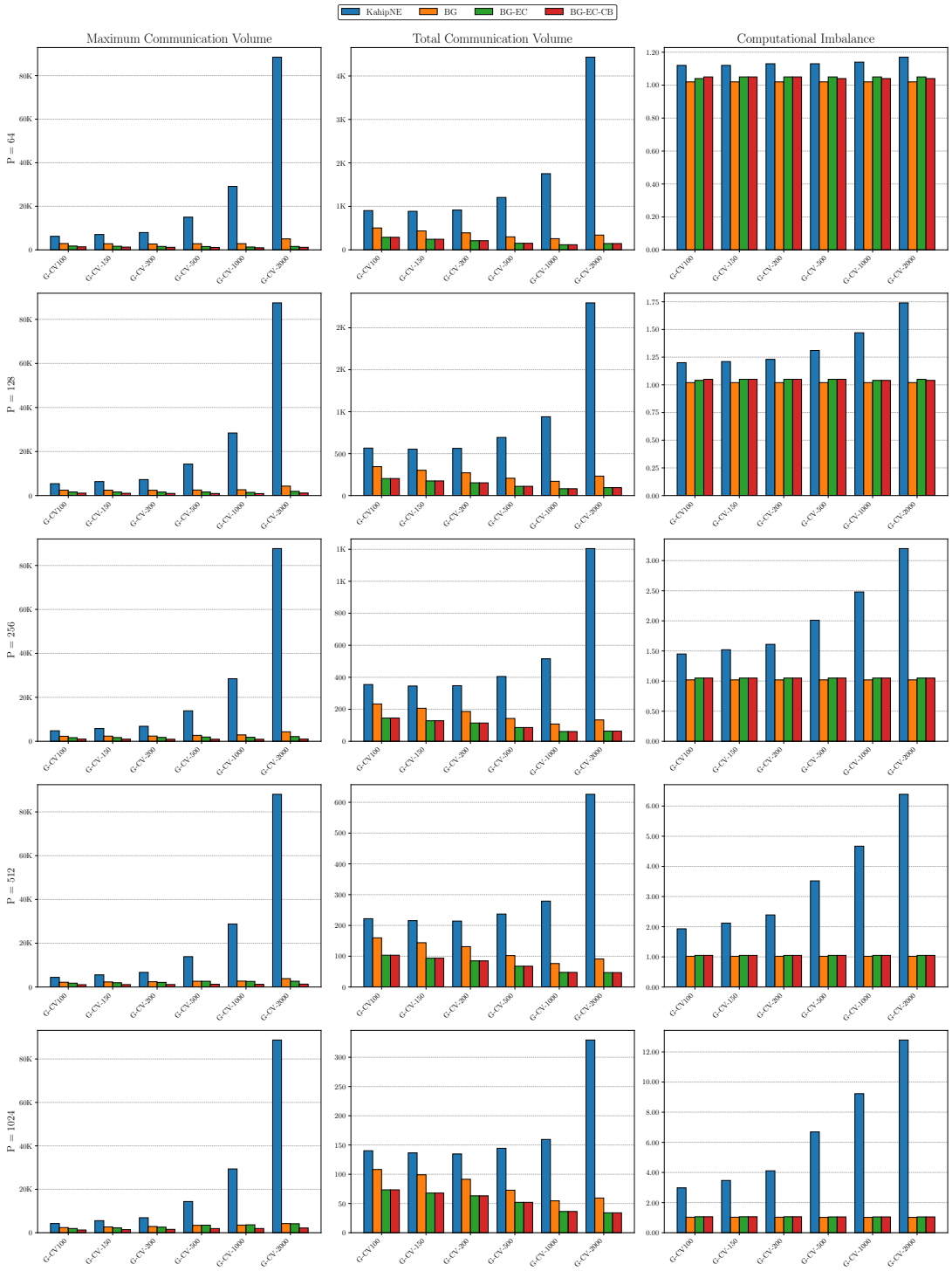


Figure 7.2: Comparison of proposed `BG`, `BG-EC`, and `BG-EC-CB` schemes against the `KahipNE` in terms of maximum volume, total volume, and computational imbalance on `ds-highcv` for $P \in \{64, 128, 256, 512, 1024\}$.

Table 7.4: Comparison of partitioning times averaged over 212 matrices (ds-highsp).

	Actual (ms)				Normalized w.r.t. 1D		
P	1D	BG	BG-EC	BG-EC-CB	BG	BG-EC	BG-EC-CB
64	252	876	641	765	3.47	2.54	3.03
128	477	1216	907	1024	2.55	1.90	2.14
256	752	1645	1310	1442	2.19	1.74	1.92
512	1204	2326	2093	2176	1.93	1.74	1.81
1024	1851	3714	3251	3514	2.01	1.76	1.90

As clearly illustrated in the graph in Figure 7.2, the proposed models consistently outperform KahipNE across all process counts and evaluation metrics. Although KahipNE attempts to achieve both edge and vertex balance, its reliance on modifying the weighting scheme, while still adhering to a coarse-grained partitioning approach, prevents it from overcoming the challenges posed by high-degree vertices. As a result, it suffers from similar issues observed in the 1D scheme, including high maximum receive volume and significant computational imbalance. This problem becomes increasingly evident with higher process counts and higher levels of scale-freeness, i.e., as the *cv* increases.

Table 7.4 displays the partitioning times of the compared schemes. For each $P \in \{64, 128, 256, 512, 1024\}$, we present the actual time and the normalized time (with respect to 1D), which are both geometric averages of 212 matrices of ds-highsp. As expected, the proposed schemes are clearly slower than the 1D scheme. The primary reason for this is that they rely on more fine-grained partitioning methods, which require partitioning a larger graph model and include additional pre- and post-processing steps. However, this performance gap narrows as the number of partitions increases. Specifically, for 64 partitions, the BG, BG-EC, and BG-EC-CB schemes are 3.47×, 2.54×, and 3.03× slower than 1D, respectively, whereas at 1024 partitions, these slowdowns decrease to 2.01×, 1.76×, and 1.90×. This is because the 1D model increasingly struggles to maintain computational balance at higher partition counts, and the relative cost of pre/post-processing becomes less significant as the total partitioning time increases. When comparing the proposed schemes among themselves, as discussed in Section 5.2,

BG-EC achieves significantly faster partitioning times than the more fine-grained BG, primarily because it reduces the graph size substantially in terms of both vertices and edges.

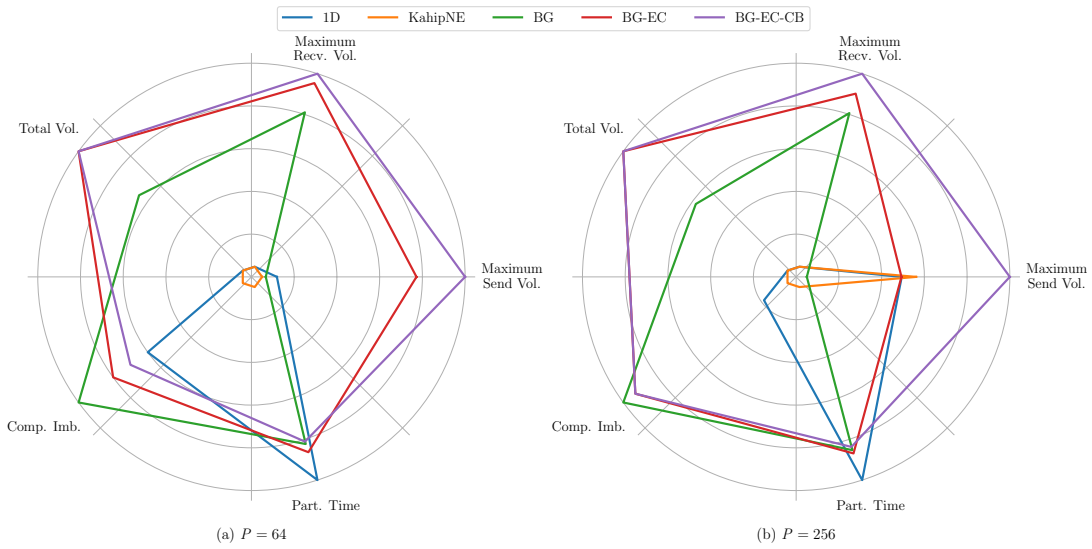


Figure 7.3: Overall comparison of proposed BG, BG-EC, BG-EC-CB schemes and against the baselines 1D and KahipNE on ds-highcv for $P \in \{64, 256\}$. (each circle represents a 20% percent improvement)

In Figure 7.3, we present a comprehensive comparison of all evaluated schemes across all considered metrics. As can be clearly seen, the proposed BG-EC and BG-EC-CB schemes consistently achieve the best performance in all metrics except partitioning time. The BG scheme, on the other hand, shows limited improvement in maximum send volume but delivers the best results in terms of computational imbalance.

Judging from the partitioning results, the experimental results demonstrate the effectiveness of the proposed bipartite-based partitioning schemes in significantly improving communication efficiency and computational balance, particularly on scale-free and sparse graphs. While they incur higher partitioning costs than the traditional 1D model, their performance gains, especially at large processor counts, highlight their suitability for scalable distributed GCN training. Among the evaluated schemes, KahipNE stands out as the weakest performer across all

metrics. Therefore, we exclude it from the parallel scalability experiments and proceed with the evaluation using the `1D`, `BG`, `BG-EC`, and `BG-EC-CB` models.

7.3 Parallel GCN runtime results

We have run parallel GCN on a MareNostrum5 machine for $P \in \{64, 128, ..., 4096\}$ processes, and on a LUMI-C machine for $P \in \{64, 128, ..., 32768\}$ processes. Due to the quota limitations on our core-hours on these systems, we have tested the performance of `BG`, `BG-EC`, and `BG-EC-CB` over `1D` for 22 test matrices included in `ds-general`. Whenever we use the phrase "parallel GCN with `BG/BG-EC/BG-EC-CB/1D`", we refer to the parallel GCN execution when the graphs are partitioned with `BG/BG-EC/BG-EC-CB/1D` schemes.

We present the results of the strong scaling experiments as the variation of parallel runtimes with increasing number of processes/cores in Figures 7.4 and 7.5. Figure 7.4 shows the strong scaling curves for the 16 graphs on MareNostrum5. Figure 7.5 displays the strong scaling curves for the 4 large graphs on LUMI-C up to 32K processors.

As seen in Figure 7.4, on MareNostrum5, the proposed schemes perform much better scaling than the conventional `1D` scheme. Out of the 16 graphs evaluated, the `1D` scheme outscales the `BG` scheme in only 6 cases. However, it is important to note that the `1D` scheme involves a single-stage communication and computation phase. Although latency overhead due to high message volumes is disregarded in this study, the two-stage communication design of the edge-vertex-parallel models may have an impact on parallel runtimes, particularly in scenarios where communication latency becomes a critical factor.

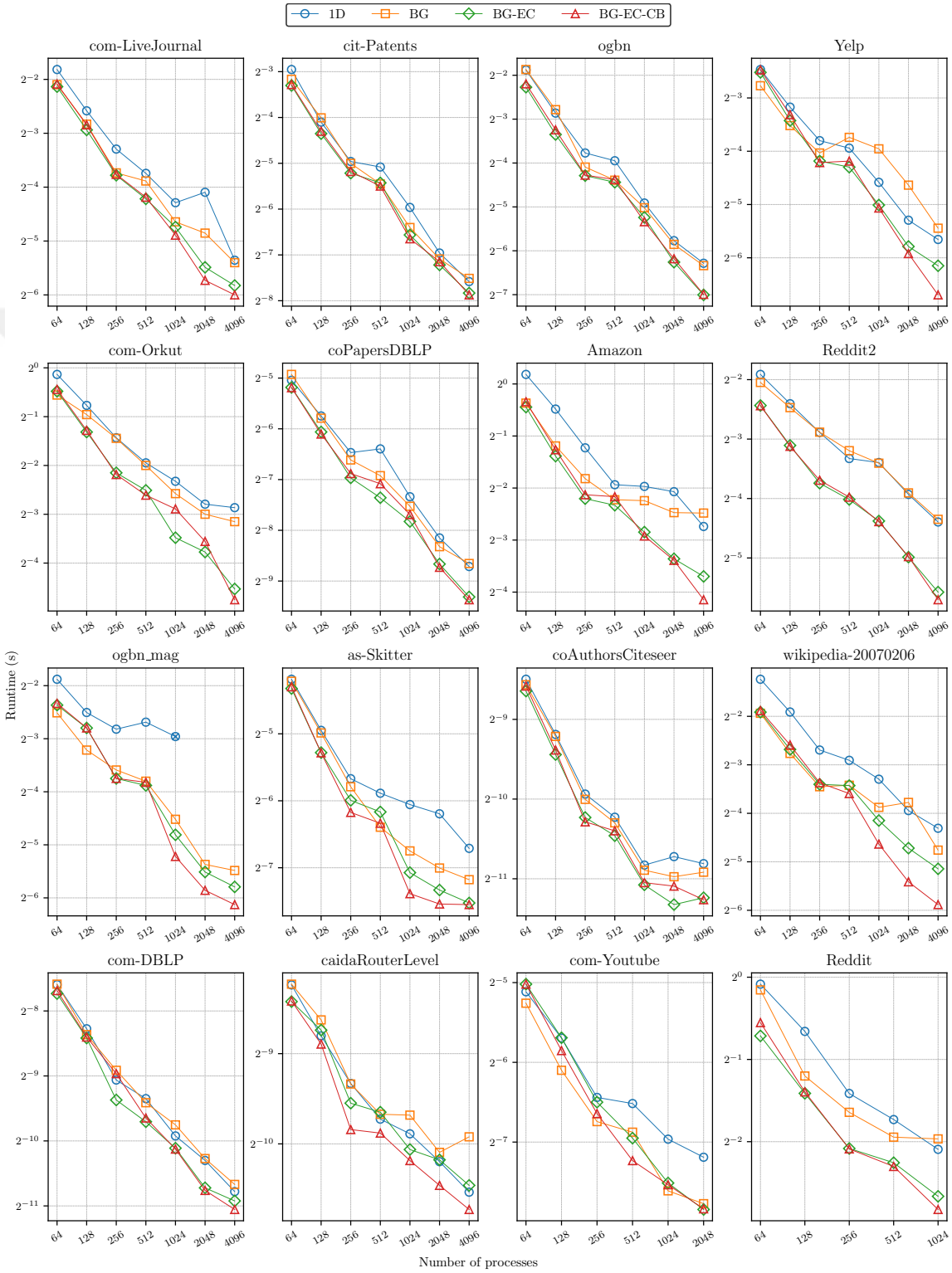


Figure 7.4: Strong scaling curves for $P \in \{64, 128, 256, 512, 1024, 2048, 4096\}$ of `1D`, `BG`, `BG-EC`, and `BG-EC-CB` schemes on MareNostrum5.

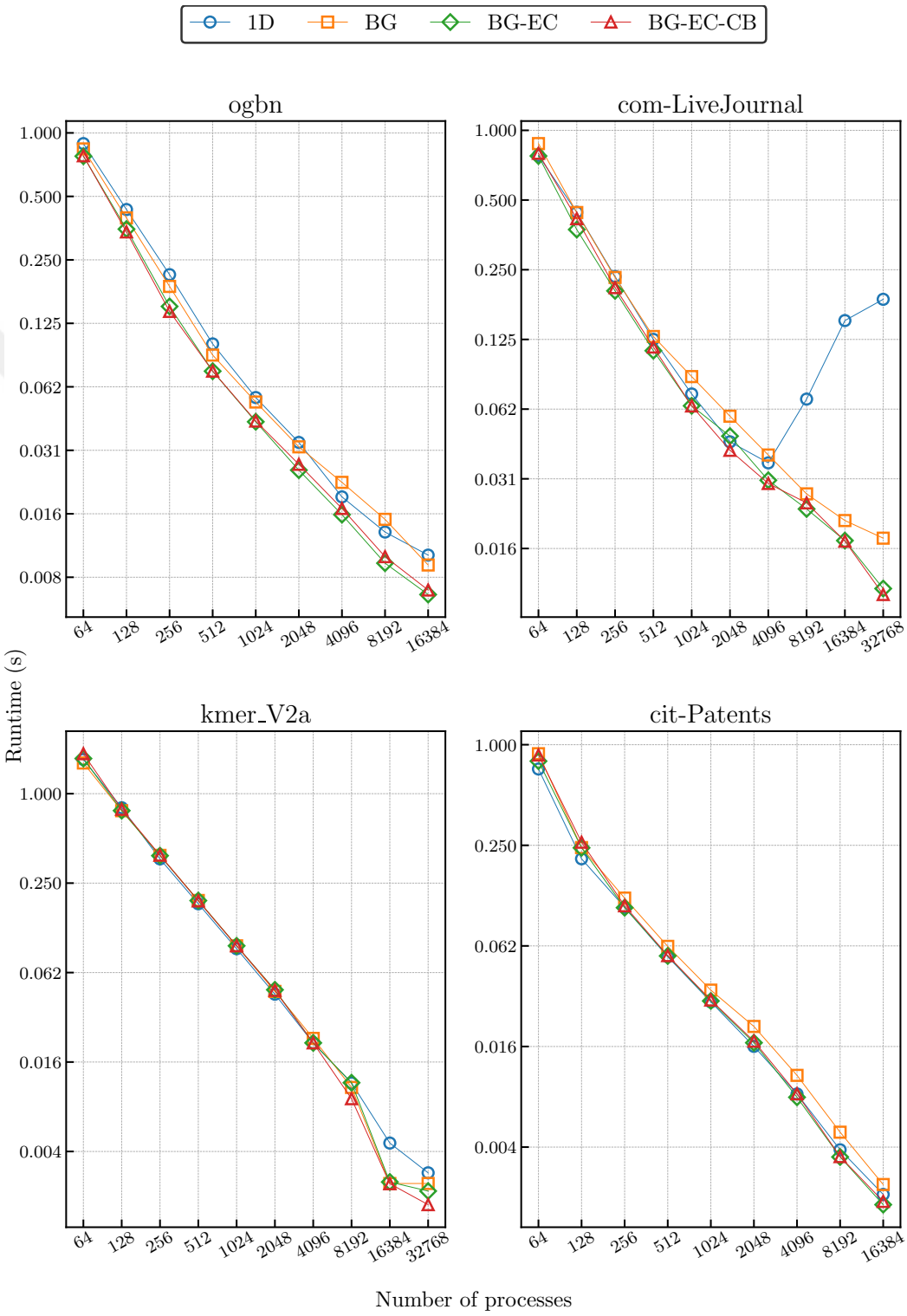
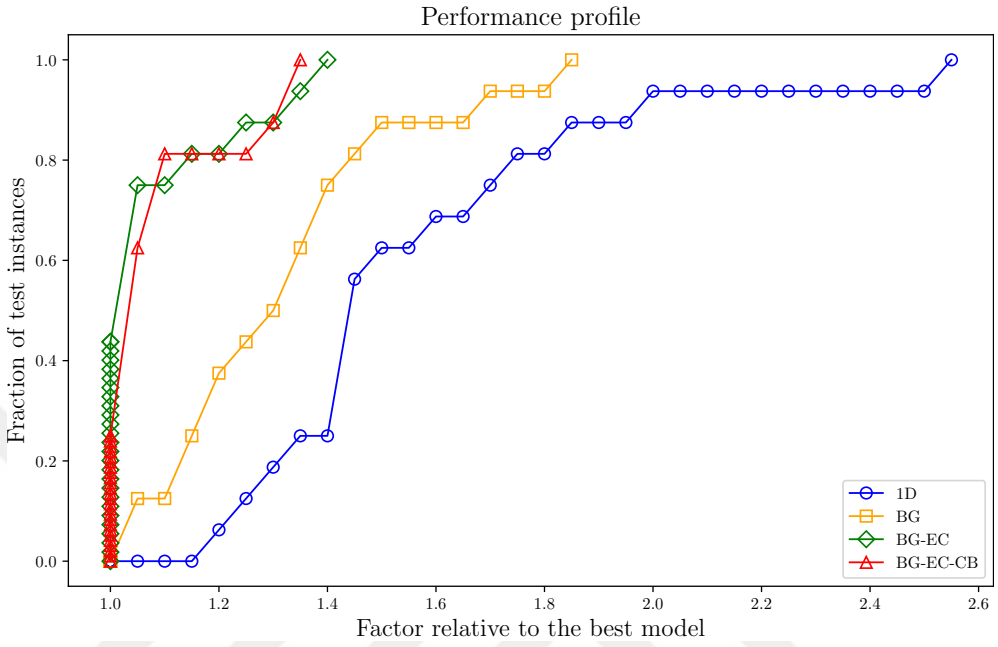


Figure 7.5: Strong scaling curves for $P \in \{64, 128, 256, ..., 16378, 32768\}$ of `1D`, `BG`, `BG-EC`, and `BG-EC-CB` schemes on LUMI-C.

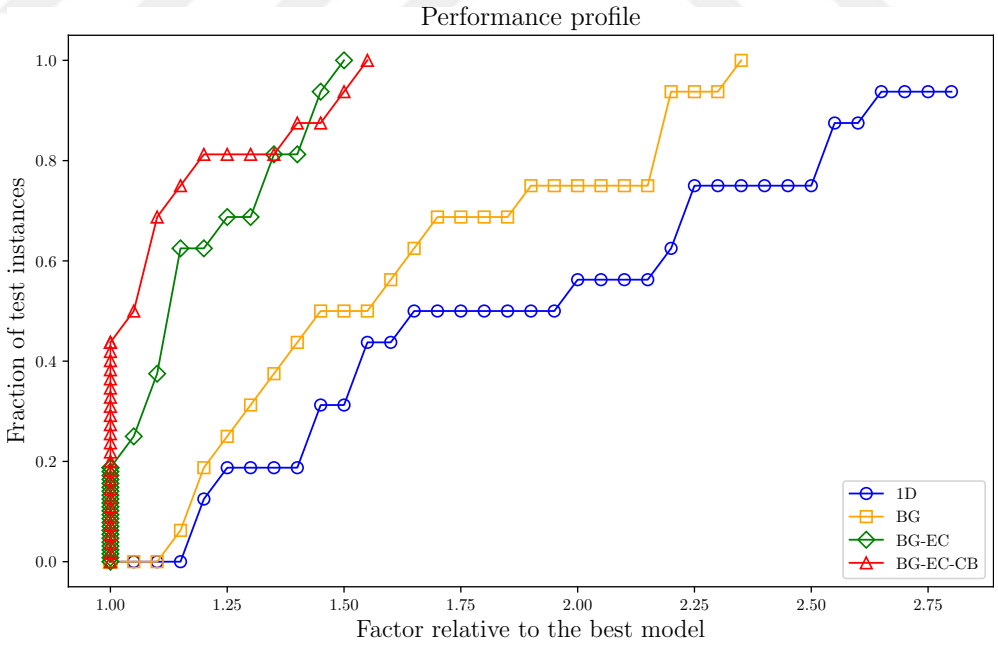
As observed in Figure 7.5, the proposed schemes `BG-EC` and `BG-EC-CB` consistently exhibit better scalability across all instances, even at high processor counts. The conventional `1D` scheme outperforms `BG` in terms of scalability in only one instance (`cit-Patents`). Notably, in the case of `com-LiveJournal`, the `1D` scheme fails to scale beyond 4096 processors, as evidenced by a sharp bend in the curve indicating a scale-down effect at higher process counts. This behavior stems from the `1D` scheme's inability to provide sufficiently balanced partitioning in terms of both communication volume and computational load at large scale. Such performance degradation is expected for large graphs under high processor counts, and the proposed schemes successfully overcome this limitation.

For a detailed comparison of the partitioning schemes in terms of parallel GCN running times/speedups, we present the runtime performance profiles in Figure 7.6. Performance profiles provide a better understanding of the characteristics of the compared models as they capture the relative performance of the compared models more accurately [50]. A point x, y in a profile reads as the respective model is within an x factor of the best result in a y fraction of the test instances. In other words, the closer the performance profile of a scheme to the y-axis, the better it is. A test instance in our case is the parallel GCN running time obtained for a specific graph and P. We compare the performances of partitioning schemes for $P = 256$ in Fig. 7.6(a) and for $P = 1024$ in Fig. 7.6(b). The former and the latter contain 22 instances.

When we compare the models for $P = 256$ in Figure 7.6(a), `BG-EC` and `BG-EC-CB` clearly emerge as the top two performing schemes, with `BG-EC` slightly outperforming the others by achieving the best performance in more than 40% of the instances. Similarly, for $P = 1024$ in Figure 7.6(b), these two schemes remain the clear leaders; however, this time, `BG-EC-CB` takes the lead by delivering the best performance in over 40% of the instances. At both processor counts, `BG` consistently ranks as the third-best model, while `1D` performs the worst, failing to achieve the best performance in any instance and showing significantly weaker performance compared to the top-performing schemes.



(a) $P = 256$



(b) $P = 1024$

Figure 7.6: Runtime performance profiles of `1D`, `BG`, `BG-EC`, and `BG-EC-CB` schemes.

In summary, the parallel runtime experiments confirm that the proposed bipartite-based partitioning schemes—particularly `BG-EC` and `BG-EC-CB`—offer superior scalability and runtime performance over the conventional `1D` model across a wide range of graphs and processor counts. Their ability to maintain balanced computation and communication even at large scales makes them well-suited for efficient distributed GCN training on modern HPC systems.

Chapter 8

Conclusion and Future Work

In this work, we proposed novel partitioning models and methods to address the scalability limitations of conventional 1D vertex-parallel approaches in distributed-memory graph neural network training. Our core contribution is a bipartite-graph-based partitioning framework that explicitly models and balances both edge and vertex computations. This bipartite formulation allows for more fine-grained control of load distribution and better alignment with the two-phase computation patterns inherent in full-batch GCN training.

To further improve partitioning efficiency and reduce the model size, we introduced a simple yet effective edge clustering method that significantly reduces the size of the bipartite graph while significantly increasing the solution quality. Additionally, we developed a communication volume balancing heuristic aimed at minimizing peak communication load, which is particularly beneficial at scale.

Our extensive experimental evaluation over hundreds of real-world, large-scale graphs demonstrated the superiority of the proposed schemes in terms of communication volume, computational balance, and parallel runtime performance. Specifically, the `BG-EC` and `BG-EC-CB` schemes consistently outperformed conventional `1D` and state-of-the-art node-edge balanced partition schemes such as

KahipNE in both partitioning quality and scalability. Parallel GCN runtime results on Tier-0 systems (MareNostrum5 and LUMI-C) confirmed that the proposed models scale more effectively than 1D, especially on graphs with scale-free and sparse structures.

As future work, we plan to extend the proposed models to support dynamic graphs and mini-batch training scenarios, which are common in large-scale, real-world GNN applications. Moreover, integrating our partitioning framework with distributed GNN training libraries such as DGL or PyG and exploring GPU-aware bipartite models for heterogeneous systems are promising directions for further performance gains.

Bibliography

[1] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs.," *SIAM J. Scient. Comput*, vol. 20, p. 359–392, 1999.

[2] F. Pellegrini and J. Roman, "Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs," in *High-Performance Computing and Networking* (H. Liddell, A. Colbrook, B. Hertzberger, and P. Sloot, eds.), (Berlin, Heidelberg), pp. 493–498, Springer Berlin Heidelberg, 1996.

[3] P. Sanders and C. Schulz, "Think Locally, Act Globally: Highly Balanced Graph Partitioning," in *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, vol. 7933 of *LNCS*, pp. 164–175, Springer, 2013.

[4] B. Hendrickson and R. Leland, "The chaco user's guide version 2.0," 07 1995.

[5] Ü. Çatalyürek and C. Aykanat, *PaToH (Partitioning Tool for Hypergraphs)*, pp. 1479–1487. Boston, MA: Springer US, 2011.

[6] G. Karypis and V. Kumar, "A parallel algorithm for multilevel graph partitioning and sparse matrix ordering," *Journal of parallel and distributed computing*, vol. 48, no. 1, pp. 71–95, 1998.

[7] F. Pellegrini, "Scotch and PT-Scotch Graph Partitioning Software: An Overview," in *Combinatorial Scientific Computing* (O. S. Uwe Naumann, ed.), pp. 373–406, Chapman and Hall/CRC, 2012.

[8] H. Meyerhenke, P. Sanders, and C. Schulz, "Parallel Graph Partitioning for Complex Networks," vol. 28, pp. 2625–2638, 2017.

[9] U. V. Çatalyürek, C. Aykanat, and B. Uçar, "On two-dimensional sparse matrix partitioning: Models, methods, and a recipe," *SIAM Journal on Scientific Computing*, vol. 32, no. 2, pp. 656–683, 2010.

[10] U. Catalyurek, "A fine-grain hypergraph model for 2d decomposition of sparse matrices," in *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*, pp. 1199–1204, 2001.

[11] E. G. Boman, K. D. Devine, and S. Rajamanickam, "Scalable matrix computations on large scale-free graphs using 2d graph partitioning," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, (New York, NY, USA), Association for Computing Machinery, 2013.

[12] A. Yoo, A. H. Baker, R. Pearce, and V. E. Henson, "A scalable eigensolver for large scale-free graphs using 2d graph partitioning," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, (New York, NY, USA), Association for Computing Machinery, 2011.

[13] B. Uçar and C. Aykanat, "Minimizing communication cost in fine-grain partitioning of sparse matrices," in *Computer and Information Sciences - ISCIS 2003* (A. Yazıcı and C. Şener, eds.), (Berlin, Heidelberg), pp. 926–933, Springer Berlin Heidelberg, 2003.

[14] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layoutl*. Chichester, U.K.: Willey–Teubne, 1990.

[15] D. J. M. Garey and L. Stockmeyer, "Some simplified npcomplete graph problems,," *Theoretical Computer Science*, vol. 1, no. 1, p. 237±267, 1976.

[16] T. N. Bui and C. Jones, "A heuristic for reducing fill-in in sparse matrix factorization," in *SIAM Conference on Parallel Processing for Scientific Computing*, 1993.

[17] B. Hendrickson and R. Leland, "A multilevel algorithm for partitioning graphs," in *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, Supercomputing '95, (New York, NY, USA), p. 28–es, Association for Computing Machinery, 1995.

[18] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2017.

[19] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," 2018.

[20] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," 2018.

[21] M. Besta and T. Hoefler, "Parallel and distributed graph neural networks: An in-depth concurrency analysis," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 46, p. 2584–2606, May 2024.

[22] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang, "Deep graph library: A graph-centric, highly-performant package for graph neural networks," 2020.

[23] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *CoRR*, vol. abs/1903.02428, 2019.

[24] J. Hu, S. Qian, Q. Fang, Y. Wang, Q. Zhao, H. Zhang, and C. Xu, "Efficient graph deep learning in tensorflow with tf_geometric," in *Proceedings of the 29th ACM International Conference on Multimedia*, MM '21, (New York, NY, USA), p. 3775–3778, Association for Computing Machinery, 2021.

[25] Y. Shao, H. Li, X. Gu, H. Yin, Y. Li, X. Miao, W. Zhang, B. Cui, and L. Chen, "Distributed graph neural network training: A survey," *ACM Comput. Surv.*, vol. 56, Apr. 2024.

[26] H. Zhou, D. Zheng, X. Song, G. Karypis, and V. Prasanna, "Disttgl: Distributed memory-based temporal graph neural network training," in *Proceedings of the International Conference for High Performance Computing,*

Networking, Storage and Analysis, SC '23, (New York, NY, USA), Association for Computing Machinery, 2023.

[27] R. Zhu, K. Zhao, H. Yang, W. Lin, C. Zhou, B. Ai, Y. Li, and J. Zhou, "Aligraph: a comprehensive graph neural network platform," *Proc. VLDB Endow.*, vol. 12, p. 2094–2105, Aug. 2019.

[28] V. Md, S. Misra, G. Ma, R. Mohanty, E. Georganas, A. Heinecke, D. Kalamkar, N. K. Ahmed, and S. Avancha, "Distgnn: scalable distributed training for large-scale graph neural networks," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, (New York, NY, USA), Association for Computing Machinery, 2021.

[29] A. Tripathy, K. Yelick, and A. Buluç, "Reducing communication in graph neural network training," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20, IEEE Press, 2020.

[30] C. Zhang, F. Wei, Q. Liu, Z. G. Tang, and Z. Li, "Graph edge partitioning via neighborhood heuristic," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, (New York, NY, USA), p. 605–614, Association for Computing Machinery, 2017.

[31] R. Chen, J. Shi, Y. Chen, B. Zang, H. Guan, and H. Chen, "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," *ACM Trans. Parallel Comput.*, vol. 5, Jan. 2019.

[32] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni, "Hdrf: Stream-based partitioning for power-law graphs," in *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, CIKM '15, (New York, NY, USA), p. 243–252, Association for Computing Machinery, 2015.

[33] N. Merkel, D. Stoll, R. Mayer, and H.-A. Jacobsen, "An experimental comparison of partitioning strategies for distributed graph neural network training," 2025.

[34] L. Ma, Z. Yang, Y. Miao, J. Xue, M. Wu, L. Zhou, and Y. Dai, "Neugraph: parallel deep neural network computation on large graphs," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, (USA), p. 443–457, USENIX Association, 2019.

[35] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, (USA), p. 17–30, USENIX Association, 2012.

[36] D. Zheng, X. Song, C. Ma, Z. Tan, Z. Ye, J. Dong, H. Xiong, Z. Zhang, and G. Karypis, "Dgl-ke: Training knowledge graph embeddings at scale," in *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '20, (New York, NY, USA), p. 739–748, Association for Computing Machinery, 2020.

[37] B. Hendrickson and T. G. Kolda, "Graph partitioning models for parallel computing," *Parallel Computing*, vol. 26, no. 12, pp. 1519–1534, 2000. Graph Partitioning and Parallel Computing.

[38] G. Karypis and V. Kumar, "Multilevel k-way partitioning scheme for irregular graphs," *Journal of Parallel and Distributed Computing*, vol. 48, no. 1, pp. 96–129, 1998.

[39] E. Horowitz, S. Sahni, and S. Rajasekaran, *Computer Algorithms*. USA: Silicon Press, 2nd ed., 2007.

[40] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, Dec. 2011.

[41] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: Datasets for machine learning on graphs," *CoRR*, vol. abs/2005.00687, 2020.

[42] K. Wang, Z. Shen, C. Huang, C.-H. Wu, Y. Dong, and A. Kanakia, "Microsoft academic graph: When experts are not enough," *Quantitative Science Studies*, vol. 1, pp. 396–413, 02 2020.

[43] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," 2013.

[44] "Yelp dataset challenge." `https://www.yelp.com/dataset`, 2020.

[45] J. McAuley, C. Targett, Q. Shi, and A. Van Den Hengel, "Image-based recommendations on styles and substitutes," *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2015.

[46] W. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *NeurIPS*, 2017.

[47] S. E. Kurt, J. Yan, A. Sukumaran-Rajam, P. Pandey, and P. Sadayappan, "Communication optimization for distributed execution of graph neural networks," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 512–523, 2023.

[48] Q. Wang, Y. Zhang, H. Wang, C. Chen, X. Zhang, and G. Yu, "Neutronstar: Distributed gnn training with hybrid dependency management," in *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, (New York, NY, USA), p. 1301–1315, Association for Computing Machinery, 2022.

[49] A. Buluç and J. R. Gilbert, "The combinatorial blas: design, implementation, and applications," *Int. J. High Perform. Comput. Appl.*, vol. 25, p. 496–509, Nov. 2011.

[50] E. D. Dolan and J. J. Moré, "Benchmarking optimization software with performance profiles," 2004.

[51] A. Pothen, H. D. Simon, and K.-P. Liou, "Partitioning sparse matrices with eigenvectors of graphs," *SIAM Journal on Matrix Analysis and Applications*, vol. 11, no. 3, pp. 430–452, 1990.

[52] C. Fiduccia and R. Mattheyses, "A linear-time heuristic for improving network partitions," in *19th Design Automation Conference*, pp. 175–181, 1982.

[53] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *The Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, 1970.

Appendix A

Multilevel Graph Partitioning

Multilevel graph partitioning (MLGP) is a widely adopted heuristic framework designed to compute high-quality partitions of large-scale graphs efficiently. The framework operates based on the observation that solving the graph partitioning problem on a smaller graph is computationally more tractable and that meaningful solutions can be projected and refined on the original graph. The process, illustrated in Figure A.1, follows a *V-cycle* structure composed of three phases: coarsening, initial partitioning, and uncoarsening with refinement.

In the *coarsening phase*, the original input graph $\mathcal{G}_0$ is successively transformed into a hierarchy of smaller graphs $\mathcal{G}_1, \mathcal{G}_2, ..., \mathcal{G}_m$ through a series of vertex contractions. In each step, a matching (a set of disjoint vertex pairs) is selected, and each pair is merged into a supernode. The goal is to reduce the number of vertices while preserving the graph's global structure. The matching can be constructed randomly or via heuristics such as Heavy-Edge Matching (HEM) or Sorted Heavy-Edge Matching (SHEM), both of which favor contracting vertex pairs connected by edges with high weights or degrees.

Once the graph becomes sufficiently small (e.g., $\mathcal{G}_4$ in Figure A.1), the *initial partitioning phase* begins. A K-way partition is computed using relatively simple algorithms such as greedy heuristics or spectral bisection [1, 51], which are efficient

on small graphs with a few thousand vertices.

Following initial partitioning, the process transitions into the *uncoarsening and refinement phase*. At each level $\mathcal{G}_i$, the partitioning solution from the coarser level $\mathcal{G}_{i+1}$ is projected (mapped) back to G_i and refined locally. As depicted on the right-hand side of Figure A.1, this projection-refinement process iteratively improves the solution. Local refinement is typically performed using algorithms such as Fiduccia–Mattheyses (FM) [52] or Kernighan–Lin (KL) [53], which adjust boundary vertices to reduce cut size while maintaining load balance. Since structural imbalances introduced at coarse levels affect large regions of the graph, even small improvements at those levels often yield significant gains when unrolled back to the fine level.

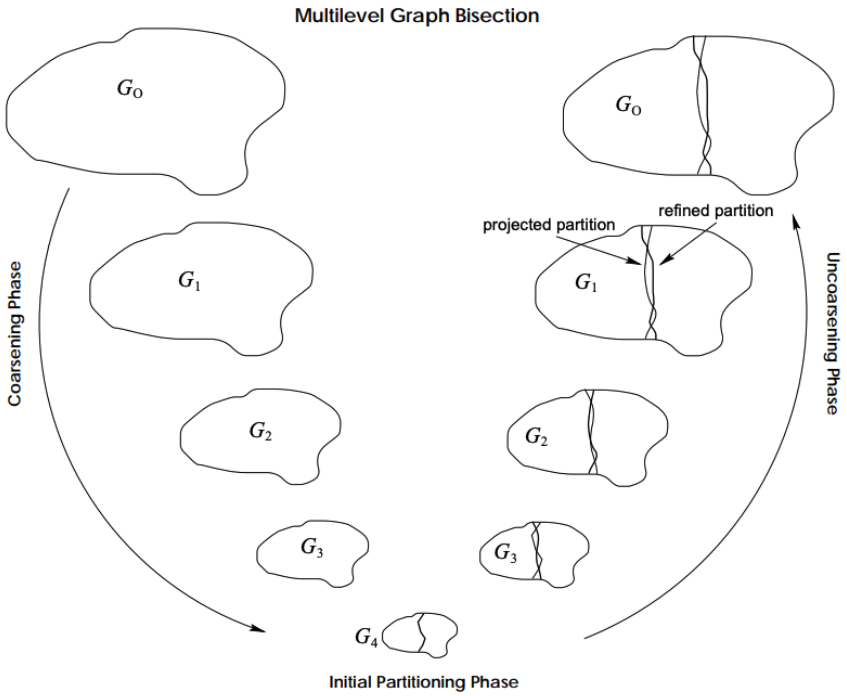


Figure A.1: Multilevel graph partitioning framework. The input graph $\mathcal{G}_0$ is successively coarsened into smaller graphs $(\mathcal{G}_1, \mathcal{G}_2, ..., \mathcal{G}_4)$, partitioned at the coarsest level, and then refined progressively during uncoarsening via projected and improved partitions [1].

This *V-cycle* procedure can be repeated multiple times, a technique known as

multi *V-cycle* refinement, to further escape local optima and improve partition quality. Overall, MLGP provides a balance between partitioning speed and solution quality and has been successfully employed in well-established tools such as Metis, ParMetis, KaHIP, and Scotch [1, 3, 6, 2]. It remains a foundational method for partitioning in applications such as VLSI design, scientific computing, sparse matrix reordering, and machine learning.