IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 5 December 2024, accepted 3 February 2025, date of publication 7 February 2025, date of current version 13 February 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3539976

RESEARCH ARTICLE

MQ-GNN: A Multi-Queue Pipelined Architecture for Scalable and Efficient GNN Training

IRFAN ULLAH[1] **AND YOUNG-KOO LEE**[2], **(Member, IEEE)**
[1]Department of Computer Science and Engineering, Kyung Hee University, Global Campus, Yongin-si 17104, Republic of Korea
[2]College of Software, Kyung Hee University, Global Campus, Yongin-si 17104, Republic of Korea

Corresponding author: Young-Koo Lee (yklee@khu.ac.kr)

This work was supported in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) grant funded by Korean Government [Ministry of Science and ICT (MSIT)] (Development of a Distributed Graph Data Management System (DBMS) for Intelligent Processing of Big Graphs) under Grant 2021-0-00859, and in part by IITP grant funded by Korean Government (MSIT) [Artificial Intelligence Convergence Innovation Human Resources Development (Kyung Hee University)] under Grant RS-2022-00155911.

ABSTRACT Graph Neural Networks (GNNs) are powerful tools for learning graph-structured data, but their scalability is hindered by inefficient mini-batch generation, data transfer bottlenecks, and costly inter-GPU synchronization. Existing training frameworks fail to overlap these stages, leading to suboptimal resource utilization. This paper proposes MQ-GNN, a multi-queue pipelined framework that maximizes training efficiency by interleaving GNN training stages and optimizing resource utilization. MQ-GNN introduces Ready-to-Update Asynchronous Consistent Model (RaCoM), which enables asynchronous gradient sharing and model updates while ensuring global consistency through adaptive periodic synchronization. Additionally, it employs global neighbor sampling with caching to reduce data transfer overhead and an adaptive queue-sizing strategy to balance computation and memory efficiency. Experiments on four large-scale datasets and ten baseline models demonstrate that MQ-GNN achieves up to $\mathbf{4.6}\times$ faster training time and $\mathbf{30\%}$ improved GPU utilization while maintaining competitive accuracy. These results establish MQ-GNN as a scalable and efficient solution for multi-GPU GNN training. The code is available at MQ-GNN.

INDEX TERMS Graph neural network, multi-GPU, pipeline, optimization, mixed CPU-GPU training, GNN training, staleness, inter-GPU communication.

I. INTRODUCTION

Graphs naturally represent complex relationships in many real-world applications, such as social networks, biological systems, and knowledge graphs [1]. By learning low-dimensional embeddings of graph nodes, GNNs efficiently represent graph information, enabling a wide range of downstream applications, such as node classification [2], [3] and link prediction [1], [4], [5]. A diverse array of models [2], [3] has demonstrated state-of-the-art performance across a variety of domains, including protein structures [6], social networks [7], and knowledge graphs [8].

Despite their success, training GNNs on large-scale graphs in a distributed GPU environment presents significant challenges due to memory constraints, computational costs, and the interdependencies inherent in graph structures. Real-world graphs often contain millions of nodes and edges, along with high-dimensional node and edge features [9], [10]. These factors result in substantial storage and computational overheads. Specific datasets may require numerous GBs of storage [11], making it inefficient or even infeasible to consider all neighbors within L hops for each training node as a single batch when training a GNN model with L layers [12]. Furthermore, most graphs exhibit highly skewed degree distributions [13], where a small number of well-connected nodes aggregate features from a substantial portion of the graph within just a few hops. These challenges necessitate mini-batch training, which reduces computational and memory overhead by processing smaller samples iteratively [14].

For mini-batch training, sampling-based approaches such as node-wise sampling [2], [15] and layer-wise sampling [4], [5], [16] have been developed. Node-wise and layer-wise

The associate editor coordinating the review of this manuscript and approving it for publication was Barbara Guidi.

sampling are among the most widely used and practical approaches for training GNNs, each addressing unique scalability and computational efficiency challenges. Node-wise sampling selects a fixed number of neighbors for each node. However, it introduces additional computational challenges, particularly when the mini-batch size increases. For instance, computing embeddings in L-layer GNNs requires aggregating information from L-hop neighbors, leading to the well-known "neighbor explosion" problem [15], [16], [17]. To address this issue, layer-wise sampling was developed. It selects subsets of nodes at each layer for sample generation. However, both node-wise and layer-wise approaches require frequent data transfers between CPU and GPU memory, reducing training efficiency. To reduce frequent and redundant data transfers, global neighbor sampling (GNS) [18] periodically samples a small number of nodes for all mini-batches and caches their features in GPU memory. GNS prioritizes neighbors in the cache, reducing distinct nodes per batch, increasing overlap among them, and minimizing CPU-GPU data transfers to speed up training. However, sample generation, data transfer, and computation do not overlap efficiently. The lack of efficient queues and pipelines and poor interleaving of GNN training tasks such as mini-batch preparation, data transfer, computation, and updates leads to significant latency and underutilized computational resources. This issue is particularly severe for large-scale graphs, where sampling and data movement dominate the runtime, causing GPUs to starve for data and stalling the training process.

Recently, multiple GPUs have been extensively used to accelerate GNN training [19], [20]. In a typical single-machine multi-GPU setup, large-scale graphs (both topology and feature data) exceed limited GPU memory capacity (e.g., 12 GB on an NVIDIA 3060), so most existing GNN systems store topological and feature data in host memory. CPUs repeatedly sample input graphs, extract features for sampled nodes, and transfer them to GPUs for training. Unfortunately, this process incurs high data transfer costs, dominating training and significantly underutilizing GPUs. This occurs because GNN models often use deep neural networks with minimal computational complexity relative to large data volumes. As a result, GNN computation on a GPU is substantially faster than data loading. For instance, [12] reported that data loading consumed 74% of training time. Existing GNN training frameworks, such as Deep Graph Library (DGL) [21] and PyTorch Geometric (PyG) [22], partially interleave processes such as mini-batch generation, data transfer, and computation. However, these frameworks still exhibit significant inefficiencies, as the interleaving cannot overlap these tasks fully. GPUs frequently become idle while waiting for the CPU to complete sampling and data transfer, resulting in poor computational resource utilization (up to 50%). These bottlenecks are aggravated in larger graphs, leading to redundant accesses to nodes, increased communication costs, and increased training time. In multi-GPU systems, the demand for data samples increases linearly, increasing communication and synchronization overheads and further limiting scalability. The system must generate and distribute more mini-batches to keep all GPUs active as the GPU count increases. This increases the system's burden, including generating samples, fetching CPU features, and moving them to GPUs. The communication volume also grows with more GPUs, leading to more significant overheads and delays. Each GPU needs to exchange gradients or intermediate results with other GPUs to update the models. Coordinating model updates and gradients across GPUs requires frequent synchronization. As the number of GPUs increases, the cost of this synchronization rises, resulting in bottlenecks where GPUs sit idle while others catch up.

To overcome these limitations, we propose MQ-GNN, a multi-queue pipelined architecture. This versatile and scalable pipeline architecture optimizes resource utilization by interleaving mini-batch generation, data transfer, GNN computation, gradient sharing, and model updating. MQ-GNN achieves this by employing multi-queues to manage mini-batches in CPU and GPU memory, along with gradients in GPU memory. By leveraging mini-batch queues in the main memory (one per GPU), MQ-GNN ensures concurrent data transfer and computation, minimizing GPU idle times. It is designed to work efficiently with any node-wise or layer-wise sampling approach, providing robust training performance regardless of the sampling method. Additionally, MQ-GNN employs global neighbor sampling to periodically cache shared nodes across mini-batches, reducing redundant accesses and CPU-GPU data transfers to accelerate training.

Furthermore, MQ-GNN introduces the Ready-to-Update Asynchronous Consistent Model (RaCoM), which uses gradient queues to enable asynchronous gradient sharing between GPUs. This allows gradients to be accumulated while models are updated concurrently with the next mini-batch computation. MQ-GNN employs periodic synchronization with intervals adapted to dataset sparsity or density to ensure model consistency and mitigate divergence. This approach balances asynchronous gradient sharing and updates, which decreases staleness and improves training efficiency.

Determining the optimal queue size in a multi-queue pipeline is crucial for balancing computational efficiency and memory usage. A large queue inflates memory usage and may cause overflows, whereas a small queue increases latency and underutilizes resources. This trade-off becomes critical when training on large datasets with diverse memory and computational demands. We propose a method to determine the optimal queue size by (1) integrating sampling and data transfer time with GPU compute time and (2) capping it at peak memory usage. This ensures feasibility for memory-intensive datasets while maximizing GPU utilization within memory constraints.

Finally, we investigated both node-wise and layer-wise sampling approaches to show MQ-GNN's effectiveness and adaptability to different sampling approaches in GNN training. Furthermore, we conducted a comprehensive multi-GPU analysis of both sampling approaches, examining how GPU count, dataset variability, and staleness affect performance. To the best of our knowledge, this is the first work that assesses layer-wise sampling in multi-GPU training within a pipelined, multi-queued framework, demonstrating MQ-GNN's scalability and efficiency across diverse configurations.

In summary, the key contributions of this paper are:

1) Proposing MQ-GNN, a multi-queue pipelined architecture, optimizes CPU, memory, and GPU utilization by interleaving mini-batch generation, data transfer, computation, and gradient sharing. This approach reduces data movement latency and maximizes GPU efficiency during GNN training.
   - To improve GPU utilization, a method to determine the optimal queue size based on available memory and processing time is introduced.
   - Employing global neighborhood sampling to cache frequently accessed nodes, reducing redundant accesses and minimizing CPU-GPU data transfers.
2) Proposing RaCoM, a framework for asynchronous gradient sharing that uses gradient queues to enable independent local model updates on GPUs, reducing synchronization delays.
   - Introducing a periodic synchronization mechanism to improve model consistency by balancing communication overhead and gradient staleness based on graph sparsity or density, enhancing scalability and robustness.
3) Conducting a comprehensive empirical analysis of node-wise and layer-wise sampling in multi-GPU settings, providing the first evaluation of layer-wise sampling performance and staleness in a multi-GPU setup, offering insights into scalability and efficiency.

II. RELATED WORK

The growing scale of graph datasets has led to the development of efficient and scalable training paradigms for Graph Neural Networks (GNNs). This section is devoted to discussing and reviewing the existing literature.

A. SCALABLE GNN TRAINING

Several systems have been designed to facilitate GPU-based GNN training [18], [23], [24]. Frameworks such as Deep Graph Library (DGL) [21] and PyTorch Geometric (PyG) [22] utilize CPU memory for graph storage and enable distributed mini-batch training across multiple GPUs. However, these frameworks often face inefficiencies, including GPU underutilization, sub-linear speedups in multi-GPU setups [25], and high data transfer overhead. To overcome these challenges, PaGraph [12] minimizes data transfer overhead with computation-aware caching, while DSP [26] improves multi-GPU training through dynamic graph partitioning and synchronized mini-batches. MSPipe [27] optimizes temporal GNNs using pipelined execution and memory-efficient caching to mitigate staleness.

MQ-GNN is different in that it incorporates mini-batch generation, data transfer, computation, and gradient sharing into a pipelined architecture with multi-queues.

B. GRAPH SAMPLING METHODS

Graph sampling methods are crucial for addressing challenges such as neighbor explosion and memory constraints in GNN training. Node-wise sampling (NS) approaches, such as fixed neighbor sampling in GraphSAGE [2], select neighbors independently for each node, reducing computation but introducing redundancy in embedding calculations [16]. Other studies propose layer-wise (LS) sampling approaches, such as FastGCN [5]. FastGCN samples a specified number of nodes per layer based on probabilities derived from each node's degree. GNN generalization and training performance is affected because sampled nodes in consecutive layers may not be connected, as their sampling probabilities are computed independently. To ensure quick and smooth convergence, the sampling probability should ideally be computed to lower the estimation variance in FastGCN [5]. As a result, the adjacency matrix can become highly sparse or contain all-zero rows, leading to disconnected nodes and an imprecise computation graph, ultimately degrading FastGCN's training and generalization performance. Huang et al. [16] presented an adaptive and trainable sampling approach that conducts LS conditioned on the former layer to capture the inter-layer correlation and lower the estimation variance. It achieved higher accuracy than FastGCN at the cost of using a much more complicated sampling approach. Zou et al. [4] proposed LAyer-Dependent Importance Sampling (LADIES), an efficient sampling algorithm that builds on the strengths and weaknesses of previous approaches. LADIES further reduces training variance and aims to mitigate sparse connections in FastGCN. To improve scalability in sampling-based GCN training, Chen et al. [28] focuses on history-oblivious LS methods (e.g., FastGCN and LADIES), which construct sampling probabilities without relying on historical data. They revisit this approach from a matrix approximation perspective and address two key issues: suboptimal sampling probabilities and estimation biases caused by sampling without replacement in existing LS methods.

MQ-GNN integrates these advanced sampling techniques with global neighbor sampling, periodically caching frequently accessed nodes across mini-batches. This reduces redundant data access, alleviates memory bottlenecks, and accelerates training while maintaining high accuracy. It supports NS and LS strategies, ensuring efficient training regardless of the sampling method.

C. PIPELINE-BASED ARCHITECTURES

Pipelined architectures effectively improve GPU utilization by overlapping data movement and computation. Marius [29] and MSPipe [27] demonstrate the benefits of pipelining for graph embeddings and temporal GNNs, respectively. However, these systems primarily focus on small graphs or specialized application domains, limiting their scalability. DSP [26] optimizes pipelined execution for multi-GPU setups with synchronized training but depends heavily on graph partitioning for performance. Marius [29] employs queues for mini-batches but lacks support for GNN training, as it is designed for non-GNN graph embedding.

Communication overhead is a significant bottleneck in multi-GPU GNN training. GNNPipe [30] and PipeGCN [31] tackle this issue in full-batch training scenarios. GNNPipe addresses this by introducing layer-level model parallelism, partitioning GNN layers across GPUs to reduce communication volume proportionally to the number of layers. Each GPU processes the entire graph for its assigned layers, utilizing historical embeddings and specialized training techniques to ensure convergence. While GNNPipe reduces communication and training overhead, its reliance on full-graph processing limits scalability for large graphs. Its reliance on layer partitioning also reduces flexibility for integrating sampling-based methods. PipeGCN reduces inter-partition communication overhead by overlapping communication and computation. In contrast, MQ-GNN eliminates full-graph dependencies through a sampling-based framework. Its pipelined architecture minimizes communication overhead without full-graph processing. MQ-GNN adapts to diverse datasets and computational setups by supporting node- and layer-wise sampling.

MQ-GNN introduces a multi-queue-based pipeline for large-scale GNN training without partitioning the graph. It interleaves mini-batch generation, data transfer, and gradient sharing, improving efficiency. Dedicated queues for mini-batches and gradients mitigate staleness, while periodic synchronization ensures consistency. This design enables efficient GNN training on both dense and sparse graphs, addressing bottlenecks in data movement, computation, and inter-GPU communication.

III. PRELIMINARIES AND NOTATIONS

This section introduces the necessary notation and background for various sampling methods in GNN training.

Let $\mathbf{G} = (V, E)$ be a graph. The node set is defined as $V = \{v_i \mid i \in [n]\}$, where $[n] = \{0, 1, 2, \ldots, n-1\}$ represents the index set of nodes. The edge set is given by $E = \{(v_i, v_j) \mid i, j \in [n]\}$. An edge $(v_i, v_j) \in E$ represents a connection between nodes v_i and v_j. Each node v_i has a feature vector $f_i \in \mathbb{R}^d$, where d is the feature dimension. All node feature sets are denoted as $F = \{f_i \mid v_i \in V\}$.

Many sampling methods can be employed during GNN training. Sampling is very important for creating mini-batches, which enhances training throughput and accelerates model convergence. These sampling methods, based on the GNN layer and node neighborhood, are as follows:

A. NODE-WISE SAMPLING (NS)

NS selects a subset of neighbors for each node to construct smaller, computationally manageable subgraphs for training. Its primary goal is to reduce memory and computational costs while preserving the graph's local neighborhood structure.

GCN For a given graph $\mathbf{G}$, the l-th convolution layer in GCN can be defined as:

$$Z^l = PH^{l-1}W^{l-1}, H^{l-1} = \sigma(Z^{l-1}), \tag{1}$$

where L is the total number of layers, $l \in \{1, \ldots, L\}$ is the layer index, σ is an activation function, H^{l-1} represents the embedding at layer $l-1$, Z^{l-1} is the intermediate embedding at layer $l-1$, and W^{l-1} denotes the weight matrix. P is the normalized Laplacian matrix, defined as:

$$P = \widehat{D}^{-1/2}\widehat{A}\widehat{D}^{-1/2}, \widehat{D}_{i,j} = \sum_j \widehat{A}_{i,j}, \tag{2}$$

where $\widehat{D}$ is a diagonal matrix, and $\widehat{A}$ represents a normalization of adjacency matrix A, i.e., $\widehat{A} = A + I$.

GraphSAGE GraphSAGE [2] uses neighbor sampling to control the receptive field size in GNNs. For each node at the l-th layer, a fixed number of its neighbors, denoted as s_{node}, are randomly and uniformly sampled. This approach formulates an unbiased estimator, $\widehat{P}^{l-1}H^{l-1}$, to approximate PH^{l-1} in the graph convolution layer and optimize computation and memory efficiency [28].

$$\widehat{P}^{l-1}_{i,j} = \begin{cases} \frac{|\mathcal{N}(v_i)|}{s_{node}} P_{i,j}, & \text{if } v_j \in \widehat{\mathcal{N}}^{l-1}(v_i) \\ 0, & \text{otherwise,} \end{cases} \tag{3}$$

where $\mathcal{N}(v_i)$ and $\widehat{\mathcal{N}}^{l-1}(v_i)$ denote the full and sampled neighbor sets of node v_i at the $(l-1)-th$ layer.

B. LAYER-WISE SAMPLING (LS)

Instead of sampling a fixed set of neighbors for each node, LS selects a globally defined set of nodes at each layer. LS offers broader receptive field coverage and reduces variation in node representations compared to NS. By adopting a layer-wide perspective, LS balances computational cost and information coverage, making it well-suited for large graphs and tasks requiring robust layer-level feature aggregation. However, LS may be less effective for tasks that depend on preserving fine-grained local neighborhood structures.

FastGCN Instead of sampling neighbors for each node, FastGCN [32] selects nodes at the layer level, modeling each graph layer as an embedding function over nodes, which are treated as random variables under a probability measure P. Importance sampling probabilities determine how nodes are prioritized for selection during training, reducing the variance in training outcomes. Each node i is assigned a probability $p_i \propto \|RP_i\|^2$ for $i \in [n]$, where R is the row selection matrix, assuming independence. For a given layer l, r_l i.i.d. samples $\{u^l_1, \ldots, u^l_{r_l}\} \sim P$ are drawn to approximate the embeddings for all nodes in the layer, using the following formulation [5], [32]:

$$z^{l+1}_{r_{l+1}}(v) = \frac{1}{r_l}\sum_{i=1}^{r_l} \widehat{A}\left(v, u^l_i\right) z^l_{r_l}\left(u^l_i\right) W^l,$$

$$h_{r_l+1}^{l+1}\left(v\right)=\sigma\left(\widehat{z}_{r_{l+1}}^{l+1}\left(v\right)\right),\quad l=0,1,\ldots,L-1, \tag{4}$$

where $h_{r_l+1}^{l+1}$ represents the embedding at layer $l+1$, $z_{r_0}^0=z^0$ denotes the node features (f), $\widehat{A}\left(v,u_i^l\right)$ corresponds to the (v,u) element of $\widehat{A}$. The loss function with function g can be estimated as:

$$\text{loss}_{r_0,r_2,\ldots,r_L}=\frac{1}{r_L}\sum_{i=1}^{r_L}g\left(z_{r_L}^L(u_i^{r_L})\right). \tag{5}$$

LADIES Independently performing layer-wise sampling at different layers can be inefficient, as the resulting bipartite graph may be sparse or even contain all-zero rows. LADIES [4] modifies independent layer-wise sampling to construct the computation graph in FastGCN training to address this issue. In their approach, at each layer, nodes are sampled from the union of the neighbors of the previously sampled nodes, as given by

$$\mathcal{V}^{l-1}=\bigcup_{v_i\in\mathcal{S}_l}\mathcal{N}(v_i), \tag{6}$$

where $\mathcal{V}^{l-1}$ represents the sampled nodes at layer $l-1$, $\mathcal{S}_l$ is the set of nodes sampled at the l-th layer, and $\mathcal{N}(v_i)$ denotes the set of neighbors of node v_i. Therefore, during the sampling process, probabilities are assigned only to nodes in $\mathcal{V}^{l-1}$, given by $\left\{p_i^{l-1}\right\}_{v_i\in\mathcal{V}^{l-1}}$. Specifically, the selection probabilities for nodes are defined as:

$$p_i^{l-1}=\frac{\left\|R^lP_{*,i}\right\|_2^2}{\left\|R^lP\right\|_F^2}, \tag{7}$$

where R^l is the row selection matrix. Assuming that the sets of sampled nodes are determined at layers l and $l-2$, and that each node v_i is assigned probabilities $p_1^{l-1}\ldots p_{|V|}^{l-1}$, the diagonal matrix $S_{s,s}^{l-1}$ is defined as:

$$S_{s,s}^{l-1}=\begin{cases}\dfrac{1}{s_{l-1}p_{i_k^{l-1}}^{l-1}}, & s=i_k^{l-1},\\ 0, & \text{otherwise.}\end{cases} \tag{8}$$

Here s_{l-1} is the number of sampled nodes at layer $l-1$. Due to the absence of information on the intermediate embedding or activation matrix during the characterization of the samples at the $1-th$ layer, an essential sampling scheme is employed, relying solely on the row selection matrices R^l and a normalized Laplacian matrix L. The row selection matrix R^l can be defined as:

$$R_{c,r}^l=\begin{cases}1, & (c,r)=(c,i_c^l),\\ 0, & \text{otherwise.}\end{cases} \tag{9}$$

where i_c^l represents the sets of nodes at S_l. Meanwhile, L is defined as, $L=\widehat{D}^{-1/2}\widehat{A}\widehat{D}^{-1/2}$. Finally, the sampled training mini-batch is defined in terms of the Laplacian adjacency matrix and corresponding embeddings (or features at layer 0) as $\frac{1}{p_k^{l-1}}\widehat{L}_{*,i}$ and $\widehat{Z}_{k,*}^{l-1}$, where $\widehat{L}\leftarrow D_{\widehat{p}^l}^{l-1}\widehat{p}^l$ is the normalized Laplacian matrix, avoiding vanishing and exploding gradients.

Calibrate and Debias LS The efficiency of LS heavily depends on the importance of the sampling procedure, which estimates node aggregations using significantly fewer nodes while preserving accuracy. The selection of sampling probabilities is crucial, as it directly impacts the accuracy of GCNs [28]. To improve the scalability of sampling-based GCN training, Chen et al. [28] investigated history-oblivious layer-wise sampling techniques, such as FastGCN and LADIES, which construct sampling probabilities without relying on historical data. By analyzing these methods from a matrix approximation perspective, they identified and addressed two key issues: suboptimal sampling probabilities and estimation biases caused by sampling without replacement.

The weak or negative correlation between $\|HW_i\|$ (where W_i is the weight matrix) and $\|P^i\|$ highlights the limitations of the proportionality assumption in FastGCN and LADIES sampling probabilities. The authors recognize the constrained prior knowledge of HW in the history-oblivious setting to address this limitation. Guided by the Principle of Maximum Entropy, they assume a uniform distribution for $\|HW_i\|$. Based on this assumption, they derive the following sampling probabilities, referred to as 'flat' sampling [28]:

$$p_i\propto\|HW_i\|,\quad\forall i\in[n]. \tag{10}$$

The sampling probabilities are adjusted to balance variance and accuracy by reformulating the target matrix product as $RPI(HW)$, where RPI is approximated. Assuming a uniform distribution for the norms of rows in HW improves both the stability of the matrix approximation and the prediction accuracy of GCNs, providing a more robust and reliable sampling framework. The advanced variants of LADIES and FastGCN that incorporate this approach are referred to as LADIES+flat (or LADIES+f) and FastGCN+flat (or FastGCN+f), respectively [28].

Although the probabilities in (10) improve performance, they assume that neighbor nodes are sampled with replacement to maintain unbiased GCN embeddings. However, in practical implementations, sampling is often performed without replacement, introducing bias because the estimator is designed for sampling with replacement. A debiasing approach was introduced to improve accuracy and mitigate this bias. With s sampled indices, FastGCN/LADIES utilize the following importance sampling estimator to approximate the target sum of matrices:

$$\frac{1}{|s_{node}|}\sum_{i\in[n]}\frac{X_{s_{node_i}}}{p_{s_{node_i}}},\quad s_{node}=\{1,2,\ldots,s\}, \tag{11}$$

$$X=RP(HW). \tag{12}$$

This estimator represents a weighted average of $RP(HW)$, and biases are introduced when the sampling is performed without replacement. To address this, the authors aim to preserve the linear form $Y_s=\sum_{i=1}^{s}\beta_iX_{s_{\text{node}}}$ during the debiasing process and develop new coefficients β_i for each $X_{s_{\text{node}}}$ to ensure that Y_s remains unbiased. Specifically, the debiasing is achieved through recursive weighted averaging as follows:

$$Y_0 := 0, \quad Y_{i+1} = (1 - \alpha_{i+1})Y_i + \alpha_{i+1}\Pi_{i+1},$$
$$\text{where } \Pi_{i+1} = \sum_{j \in S_i} X_i + \frac{X_{s_{node_{i+1}}}}{p^i_{s_{node_{i+1}}}},$$
$$\forall i = 0, 1, \ldots, |s_{node}| - 1. \tag{13}$$

Here, S_i represents the set of indices sampled using weighted random sampling [33], where $0 \le i \le |s_{node}| - 1$ and $S_0 = \emptyset$. $\alpha_1 = 1$, and α_{k+1} is a constant dependent on k. Specifically, $\alpha_{k+1} = \frac{n}{(n-k)(k+1)}$ is chosen to ensure that when all $p_i = 1/n$, the output coefficients align with those of a simple random sampling setting. This approach is referred to as 'debias' in the original paper. The LADIES and FastGCN variations incorporating this method are LADIES+debias (LADIES+d) and FastGCN+debias (FastGCN+d), respectively. Furthermore, sampling methods that incorporate both flat sampling (10) and debiasing (13) are referred to as LADIES+flat+debias (LADIES+f+d) and FastGCN+flat+debias (FastGCN+f+d), respectively.

C. FEATURE CACHE

To accelerate training, Global Neighbor Sampling (GNS) [18] facilitates efficient neighbor sampling within a mini-batch by periodically constructing a node cache, denoted as $\mathcal{C}$, where $\mathcal{C} = \{c_i \mid i \in \mathcal{V}_c \subset V\}$. The nodes in $\mathcal{C}$ are selected using a biased sampling strategy to ensure they can be reached from the training set nodes with high probability. The features of nodes in $\mathcal{C}$ are preloaded onto GPUs during training. To fit within GPU memory constraints, only 1% of the nodes that are highly likely to be reachable from the training set are cached. This strategy effectively reduces data movement overhead and speeds up training by caching a small subset of high-degree nodes. GNS defines two approaches for determining the sampling probability of the cache. If the majority of the nodes in a graph belong to the training set, the sampling probability is determined based on the node degree. For a node i, the probability of being sampled in the cache is given by:

$$p_{v_i} = \frac{\deg^-(v_i)}{\sum_{v_j \in V} \deg^-(v_j)}, \tag{14}$$

where $\deg^-(v_i)$ denotes the in-degree of node v_i. In power-law graphs, caching a small subset of nodes is sufficient to cover most nodes due to the highly skewed degree distribution. When the training set includes only a small subset of the graph's nodes, GNS performs short random walks to compute the sampling probability. Considering sampled neighbors $\mathcal{N}_v$ of $v \in V$, the sampling ratio $\mathbf{d}$ is defined as:

$$\mathbf{d} = \left\{ \frac{\mathcal{N}(v_1)}{\deg^-(v_1)}, \frac{\mathcal{N}(v_2)}{\deg^-(v_2)}, \ldots, \frac{\mathcal{N}(v_{|V|})}{\deg^-(v_{|V|})} \right\}. \tag{15}$$

The node sampling probability $P^{(l)} \in \mathbb{R}^{|V|}$ for the $l - th$ layer is computed as:

$$P^{(l)} = (\mathbf{D}A + I)\, P^{(l-1)}, \quad \mathbf{D} = \text{diag}\,(\mathbf{d})\,.$$
$$p^{(0)}_{v_i} = \begin{cases} \frac{1}{|V_t|}, & \text{if } v_i \in V_t, \\ 0, & \text{otherwise.} \end{cases} \tag{16}$$

where V_t denotes the training set. The final sampling probability for the cache is given by $P^{(L)}$.

In summary, sampling techniques such as node-wise and layer-wise and their variants, such as flat and debiased LS, help mitigate computational and memory constraints in large-scale GNN training. These methods trade off scalability and accuracy while introducing challenges such as sampling bias and increased computational overhead. The node cache strategy further enhances efficiency by leveraging global neighbor sampling to preload frequently accessed nodes into GPU memory.

IV. MULTI-QUEUE PIPELINED GNN TRAINING ARCHITECTURE

This section presents the proposed MQ-GNN architecture for training GNN models. First, we provide an overview of the system design, followed by a detailed analysis of each training stage.

Overall Design MQ-GNN introduces a pipelined architecture to optimize CPU, memory, and GPU utilization during GNN training. It employs mini-batch queues, gradient queues, and periodic synchronization to address scalability and efficiency challenges in multi-GPU environments. The pipeline architecture of MQ-GNN facilitates a smooth data flow between mini-batch generation, data transfer, and computation, as shown in Fig. 1. The architecture consists of seven stages: three dedicated to computation, three to data transfer, and one integrated stage combining computation and data transfer. The architecture uses multiple processes and threads across stages to maximize computational overlap and enable asynchronous processing. To efficiently manage data flow, we introduce three specialized queues: two for mini-batches and one for gradients. Each queue is populated using parallel processes and threads.

In the following, we provide the details of seven different stages of the MQ-GNN pipeline:

Data Loading In the initial stage, the system loads the graph's topology and features into the main memory, preparing the data for subsequent processing for GNN training.

Model Transfer Initially, the model is initialized and transferred to the GPU(s) for training. This ensures all GPUs begin with identical model parameters, enabling asynchronous processing across devices.

Mini-Batch Generation and Enqueuing In this stage, training batches are generated through sampling and enqueued into dedicated mini-batch queues on the CPU, with each GPU assigned its queue. This parallelized design

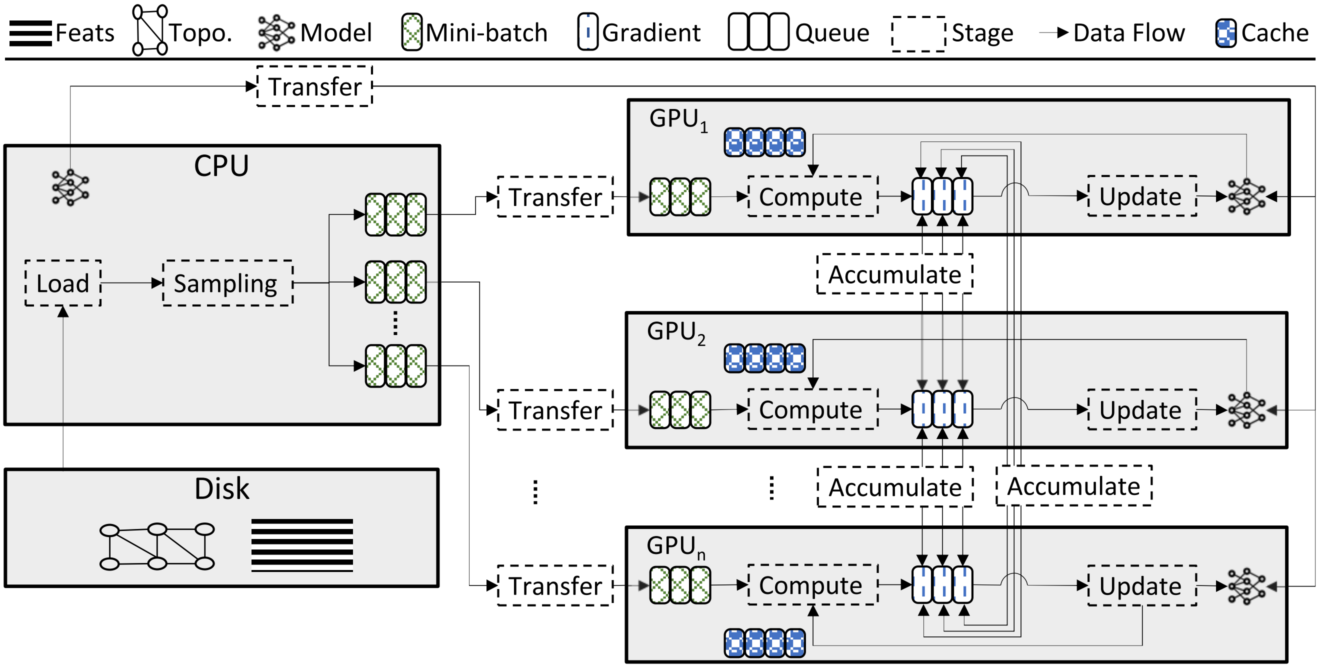

FIGURE 1. The MQ-GNN architecture integrates multi-queue pipelining to overlap mini-batch generation, data transfer, computation, asynchronous gradient sharing, and model updates across CPUs and GPUs. This design minimizes latency, maximizes GPU utilization, and accelerates large-scale GNN training.

reduces bottlenecks and enables efficient data transfer to GPUs.

Efficient mini-batch generation and management are crucial for improving training throughput, as Section III explains. To achieve this, MQ-GNN incorporates NS approaches, such as GCN [34] and GraphSAGE [2], as well as LS approaches, including FastGCN [5], LADIES [4], and their advanced variants [28], each discussed in Section III.

Mini-Batch Transfer To GPU The generated mini-batches are transferred to the mini-batch queue on the GPU(s) during the *Transfer* stage. Fig. 1 illustrates the pipeline for mini-batch generation and transfer, highlighting how computation and data movement are overlapped.

The CPU extracts features for nodes not present in the GPU cache. Missing features are gathered from the CPU's main memory and transferred to the GPU via PCIe, which can be resource-intensive. The overhead of this stage depends on the GPU cache size, cache policies, and graph properties like degree distribution. Efficient or complete GPU caching minimizes or eliminates this cost, optimizing data transfer and memory usage. Additionally, it eliminates the need for synchronization among GPUs when gathering neighboring features of target nodes on a GPU because each mini-batch on the GPU contains both features and topology components that are transferred from the CPU.

Computation The *compute* stage runs on the GPU(s), processing the enqueued mini-batches from the *Transfer* stage. This stage focuses on processing mini-batches and performing forward (Fwd) and backward (Bwd) propagation to train the GNN concurrently with the previous two stages. The *compute* stage focuses on maximizing GPU(s) utilization and optimizing compute operations by performing Fwd and Bwd operations without waiting for model updates from other GPUs. The worker thread consumes the mini-batches by performing Fwd propagation to train the GNN as follows:

$$\begin{aligned} Z^{l+1}(v) &= GNN^l\left(\left[Z^l_{agg}(v), Z^l(v)\right], W^l\right), \\ Z^l_{agg}(v) &= Mean\left(H^l(v)\right), \end{aligned} \tag{17}$$

where H^l represents the intermediate embedding at layer l, Z^l is the embedding at layer l, W^l denotes the weight matrix, and $\left[Z^l_{agg}(v), Z^l(v)\right]$ represents the concatenation of the aggregated neighborhood representation for GraphSAGE, and the previous embedding of node v at layer l.

For GCN, $Z^{l+1}(v)$ is modified to replace the concatenation with direct neighborhood aggregation as follows:

$$Z^{l+1}(v) = GNN^l\left(Z^l_{agg}(v), W^l\right) \tag{18}$$

As Fwd propagation involves two main steps: gathering and aggregating neighboring features (Gather, GA) and applying the neural network (Apply NN, AN) [35], [36]. From a matrix perspective, GA corresponds to AH matrix multiplication, while AN corresponds to $(AH)W$.

The loss for a batch s, $\mathcal{L}^s$, is calculated as:

$$\mathcal{L}^s = -\sum_{v\in V^s}\left[y_v \log(pr_v) + (1-y_v)\log(1-pr_v)\right], \tag{19}$$

where V^s denotes the set of nodes in s, y_v denotes the ground truth label, and pr_v represents the predicted probability for node v.

During Bwd propagation, the gradient computation for the GA step, denoted as ∇GA, involves AG operation, where G represents the gradient matrix. Neither GA nor ∇GA requires synchronization between GPUs.

After gradient computation, the gradients are enqueued into corresponding gradient queues rather than being synchronized immediately across GPUs. As illustrated in Fig. 2, GPU $\mathcal{G}_0$ enqueues the gradients during time τ_1 and proceeds to process another mini-batch, while the remaining GPUs ($\mathcal{G}_i$, $i \neq 0$) are still completing the first iteration. This pipelined execution facilitates optimal resource usage and overlapping operations. In MQ-GNN, gradient queues store gradients awaiting synchronization, while mini-batch queues hold training data for GPU processing. This distinction ensures a clear demarcation between the pipeline's stages.

Gradient Sharing and Accumulation Across GPUs MQ-GNN employs asynchronous gradient-sharing mechanics, accumulating gradients across GPUs to balance updates. This process reduces the impact of communication delays by ensuring that local computations proceed without interruption.

Once gradients are enqueued, they are asynchronously shared across all GPUs, ensuring prompt distribution without interrupting ongoing computations. Upon receiving delayed gradients, GPUs update their queues using the formula:

$$\Delta^{\text{New}} = \Delta^{\text{Current}} + \frac{\left(\Delta^{\mathcal{G}_i} - \Delta^{\text{Current}}\right)}{\text{Average Count}}, \tag{20}$$

where $\Delta^{\mathcal{G}_i}$ is the incoming gradient from GPU $\mathcal{G}_i$, Δ^{Current} is the existing gradient, and 'Average Count' tracks the number of gradients received (i.e., 'Average Count' $\leq |\mathcal{G}|$). This ensures balanced updates despite communication delays.

Fig. 2 illustrates gradient sharing and accumulation, GPU $\mathcal{G}_1$ enqueues and shares gradients after GPU $\mathcal{G}_0$ during τ_2, while simultaneously processing data for the next iteration. This design ensures efficient execution, preventing delays caused by synchronization or resource under-utilization.

Model Update on Each GPU Once gradients from all GPUs are received (i.e., 'Average Count' $= |\mathcal{G}|$), the accumulated gradient is computed as follows:

$$\nabla_l^{acc} = \frac{\sum_{i=1}^{|\mathcal{G}|} \nabla_l^{\mathcal{G}_i}}{|\mathcal{G}|}, \tag{21}$$

where ∇_l^{acc} is the accumulated gradient at layer l, and $\nabla_l^{\mathcal{G}_i}$ is the batch-specific gradient computed at layer l on GPU $\mathcal{G}_i$. This aggregation consolidates contributions from all GPUs processing different mini-batches. The *Update* stage applies the accumulated gradients to update model parameters on GPUs.This stage gaurantees that model parameters are updated across all GPUs in a multi-GPU environment, integrating the collective contributions of the accumulated gradients.

After completing the current iteration, each GPU incorporates the accumulated gradient into its model update before proceeding to the next iteration. The model update rule for layer l is defined as:

$$W^l = W^l - \eta \nabla_l^{acc}, \tag{22}$$

where W^l denotes the weight matrix, and η is the learning rate. Although gradient sharing facilitates efficient communication across GPUs, asynchronous updates can lead to model staleness. To mitigate this, we introduce the Ready-to-Update Asynchronous Consistent Model (RaCoM), a dynamic synchronization mechanism that ensures model consistency across GPUs.

RaCoM RaCoM addresses the challenge of staleness in asynchronous GNN training by combining asynchronous local updates with periodic synchronization, ensuring global consistency and computational efficiency. The periodic synchronization mechanism adapts to the sparsity or density of the dataset, aligning updates to balance staleness and communication costs effectively. This design ensures that the GNN model achieves scalability while preserving accuracy, even under varying workload intensities. The synchronization period P_{ms} is given by:

$$P_{ms} = \left\lceil \frac{\sqrt{|V|}}{\sqrt{|\mathcal{G}|\,|E|}} \right\rceil, \tag{23}$$

where $|V|$ and $|E|$ denote the number of nodes and edges in the graph, and $|\mathcal{G}|$ is the number of GPUs. The synchronization period P_{ms} minimizes communication overhead while ensuring timely updates. This balance ensures that RaCoM can efficiently scale to handle large graphs and high GPU counts with a slight drop in model accuracy. In MQ-GNN, periodic synchronization, managed by (23), dynamically balances synchronization costs and staleness. MQ-GNN reduces the effect of asynchronous updates on training efficiency by adapting the synchronization period P_{ms} according to the structure of the graph. This ensures optimal utilization of computational resources across GPUs while maintaining model consistency. A detailed proof of the periodic synchronization mechanism and its derivation is provided in Appendix A. These derivations highlight how P_{ms} dynamically adapts to graph properties to minimize synchronization costs.

Fig. 2 shows the mechanism of the gradient-sharing mechanism and periodic synchronization proposed in MQ-GNN. It illustrates the gradient sharing process, highlighting how worker threads overlap gradient enqueuing and computation tasks. Gradient queues are used to streamline the communication of gradients between GPUs, enabling asynchronous updates and overlapping computation. It also depicts how periodic synchronization aligns gradients across GPUs to maintain consistency. Each gradient $\Delta_t^{\mathcal{G}_i}$ generated by GPU $\mathcal{G}_i$ at iteration t is first enqueued in its local gradient queue before being propagated to all other GPUs ($\mathcal{G}_j, j \neq i$). The gradient $\Delta_t^{\mathcal{G}_i}$ incurs a delay $r_{\mathcal{G}_i,\mathcal{G}_j}^t$ before arriving at $\mathcal{G}_j$. For instance, a straggler GPU ($\mathcal{G}_n$) shares its gradient $\Delta_t^{\mathcal{G}_n}$ immediately after enqueuing, but the gradient reaches the other GPUs ($\mathcal{G}_j, j \neq n$) with a delay $r_{\mathcal{G}_j,\mathcal{G}_n}^t$. When

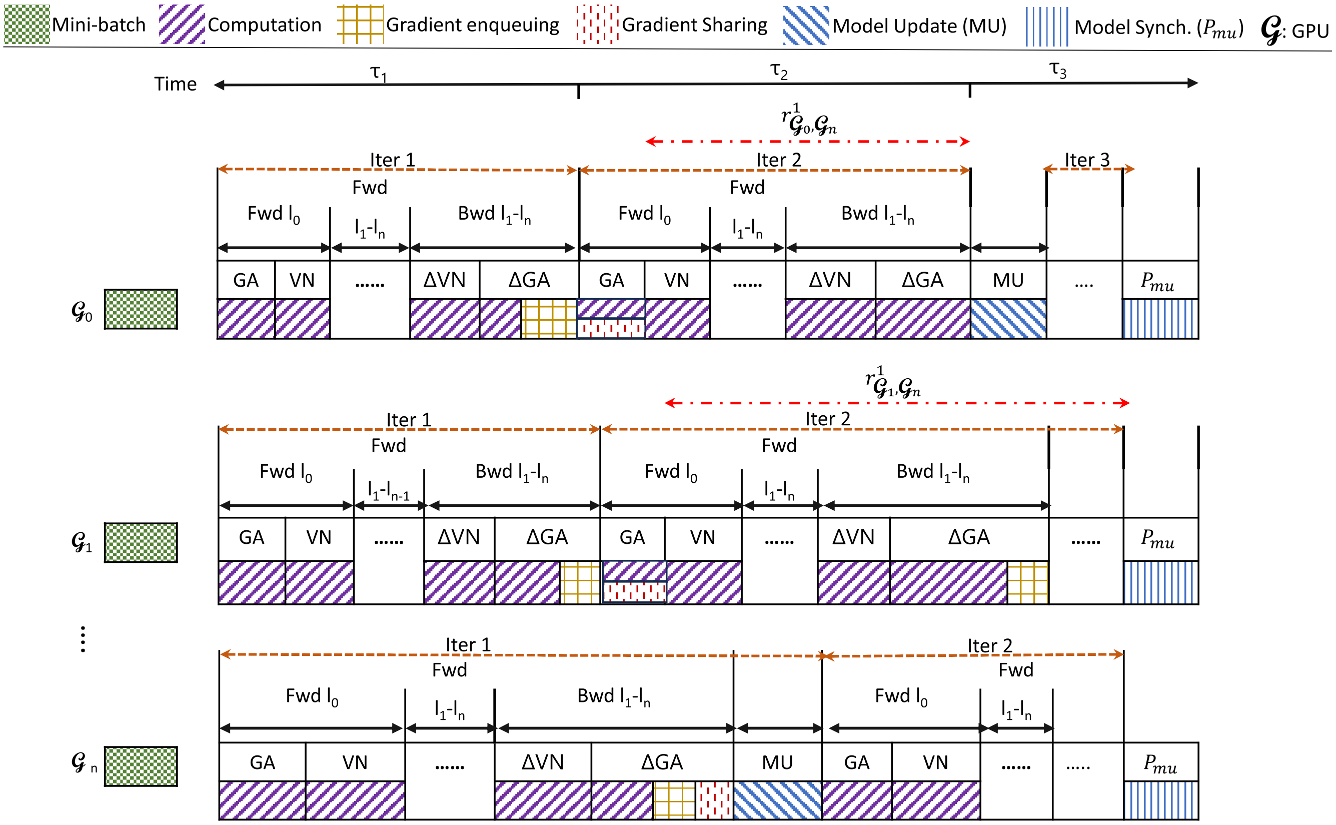

FIGURE 2. The RaCoM pipeline in MQ-GNN enables asynchronous gradient sharing, model updates, and periodic synchronization across GPUs. It efficiently overlaps mini-batch generation, data transfer, forward and backward propagation, and model synchronization P_{ms} to maximize resource utilization and minimize training latency.

the gradients from the straggler GPU arrive at a receiving GPU, they are accumulated using (20). Once the gradients (i.e., (21)) are ready to update the model, the receiving GPU completes the current iteration, incorporates the updated gradient into its model update (22), and proceeds to the next iteration. Finally, after P_{ms} iterations, model synchronization ensures consistency across all GPUs for maintaining model accuracy.

A. DETERMINING OPTIMAL QUEUE SIZE

When training GNNs on large-scale datasets, it is critical to identify the optimal queue size for efficient GPU utilization. A practical approach balances computational efficiency with memory constraints to ensure high throughput while avoiding overflows. The queue size is calculated based on batch processing times and the memory available on the GPU. This hybrid approach adjusts the queue size dynamically to maintain GPU utilization while respecting memory limits.

The queue size is given by:

$$\mathcal{Q} = \min\left(C, \max\left(2, \left\lceil \frac{\max(T_{\text{sampling}} + T_{\text{transfer}})}{\text{mean}(T_{\text{compute}})} \right\rceil\right)\right) \tag{24}$$

Here, $\max(T_{\text{sampling}} + T_{\text{transfer}})$ represents the maximum time required for sampling and data transfer. In contrast, $\text{mean}(T_{\text{compute}})$ denotes the average computation time on the GPU. The cap size, C, sets an upper bound on the queue size to prevent exceeding the GPU's memory capacity. The peak memory usage accounts for the temporary memory overheads incurred during CUDA computations. The C is defined as:

$$C = \left\lfloor \frac{\mathcal{M}_{\mathcal{Q}}}{\mathcal{M}_{\text{mini-batch}}} \right\rfloor \tag{25}$$

where $\mathcal{M}_{\mathcal{Q}}$ is the available memory for queues, and $\mathcal{M}_{\text{mini-batch}}$ is the memory per mini-batch. The $\mathcal{M}_{\mathcal{Q}}$ is calculated by subtracting the measured peak memory usage (including a safety margin of 5–10%) from the total GPU memory. The memory per batch depends on the batch size, feature dimension, and the data type used (e.g., 4 bytes for float32). This ensures that the queue size does not exceed the GPU's memory capacity while maintaining efficiency.

Peak memory usage can be estimated by running a single training epoch using profiling tools such as PyTorch Profiler or NVIDIA Nsight Systems. These tools provide helpful information regarding memory allocation and the temporary overheads of CUDA computations. Based on this peak memory value, the peak capacity can be calculated and included in the queue size formula.

TABLE 1. An overview of the datasets. One count is made for each undirected edge."Deg." stands for the graph's average degree. The ratio of training, validation, and test data is called the "split ratio."

Dataset	Nodes	Edges	Avg. Degree	Feature dimension	Classes	Split Ratio	Metric
ogbn-proteins	132,534	39,561,252	597	8	2	65/16/19	ROC-AUC
ogbn-arxiv	160,343	1,166,243	13.7	128	40	54/18/28	Accuracy
Reddit	232965	11606919	50	602	41	66/10/24	F1-score
ogbn-products	2,449,029	61,859,140	50.5	100	47	8/2/90	Accuracy

V. EVALUATION

This section comprehensively evaluates the MQ-GNN architecture by comparing it with baseline models.

A. EXPERIMENTAL SETUP

This section describes the hardware setup used for the experiments.

Hardware Setup The experiments were conducted on Linux clusters managed by the SLURM workload manager. The hardware setup in this work was determined by the maximum resource allocation permitted by the SLURM cluster account. Each job was allocated up to 4 GPUs, 32 CPUs, and 126 GB of RAM, with a maximum runtime of 4 days.

The experiments were conducted on a machine running Ubuntu 20.04.6 LTS (kernel 5.4.0-192-generic), equipped with four NVIDIA GeForce RTX 3090 GPUs (24 GB each) and 32 Intel Xeon E-2234 CPUs (3.60 GHz, four cores per socket, two threads per core). To maximize these resources, we optimized CPU, GPU, and memory use across single, two, three, and four GPU setups.

- **Experiments with a Single GPU** Each experiment used 32 CPUs and 128 GB of memory.
- **Experiments with Two GPUs** Each experiment used 16 CPUs per GPU and 64 GB of memory per GPU.
- **Experiments with Three or Four GPUs** Each experiment used 8 CPUs per GPU and 32 GB of memory per GPU.

This scaling across GPU configurations enables the evaluation of MQ-GNN's performance and scalability under different parallelism and resource contention levels.

Datasets and Evaluation Metrics To ensure fair comparisons and representative results, we conducted experiments on four widely used benchmark datasets: Reddit [2] and three Open Graph Benchmark (OGB) datasets—ogbn-arxiv, ogbn-proteins, and ogbn-products [37]. Table 1 provides a summary of these datasets. These datasets exhibit diverse properties. The Reddit dataset, widely used in prior works [5], [15], contains high-degree nodes and complex community structures, making it a standard benchmark for evaluating scalability. This dataset presents challenges due to its high average node degree, large feature dimensions, and dense subgraphs, necessitating efficient sampling strategies. The OGB datasets present additional challenges, including a class imbalance in ogbn-proteins and many nodes in ogbn-products, which test both scalability and accuracy [37]. Together, these datasets provide a comprehensive testbed for evaluating the scalability, efficiency, and generalization capabilities of MQ-GNN across diverse graph structures.

We used different evaluation metrics for each dataset, as shown in Table 1, maintaining consistency with recent literature and following the recommendations of [37] to ensure compatibility with baselines.

Models and Hyperparameters To ensure a fair comparison, all baseline models were implemented using PyTorch and DGL, maintaining consistent training parameters across all methods. For node-wise sampling, five neighbors were chosen per node, while for layer-wise sampling, 512 nodes were selected per layer. All models were trained using a two-layer GNN with the ADAM optimizer, set to a learning rate of $\eta = 0.001$. A caching strategy was employed, periodically storing 1% of the nodes per epoch to enhance efficiency. Early stopping was applied if the validation metric did not improve by at least 0.01 for 200 consecutive batches. The model with the highest validation metric was selected as the final model. Each model was trained ten times per method and dataset to ensure statistical robustness, with the mean and variance reported for evaluation. To mitigate GPU timing variability caused by kernel initialization and synchronization overheads, the first and last 20 mini-batch timings were excluded. This ensures the reported timings reflect steady-state performance, reducing noise from warm-up and cool-down phases.

B. COMPARISON WITH EXISTING SYSTEMS

To demonstrate that MQ-GNN optimizes resource utilization more effectively than existing systems, leading to faster training, we compare it against a range of baseline models. These include NS-based methods such as mini-batch GCN and GraphSAGE [21], as well as LS-based methods, including FastGCN [5], LADIES, and their advanced variants [28]. These baselines were selected based on their prominence in prior research and effectiveness in representing state-of-the-art NS-based and LS-based GNN training approaches.

The evaluation results, summarized in Tables 2 to 13, report the mean (± standard deviation) of batch and training times (in milliseconds, ms), along with evaluation metric percentages for each dataset. Table 1 summarizes dataset characteristics.

We first highlight key insights from the results before presenting a detailed discussion.

Major Highlights The MQ-GNN architecture significantly improves performance by optimizing resource utilization and reducing overhead in GNN training. Across multiple datasets, MQ-GNN achieves up to 4.6 × faster training speeds than baseline models while maintaining comparable evaluation metrics. This acceleration is primarily due to the MQ-GNN's efficient multi-queue design, which enables concurrent computation, data transfer, gradient sharing, and periodic model synchronization across GPUs. Moreover, MQ-GNN adapts effectively to various datasets and configurations, consistently achieving performance gains across various GPU setups, sampling strategies, and node cache sizes. These results underscore MQ-GNN's scalability, efficiency, and effectiveness in large-scale GNN training.

C. RESULTS AND DISCUSSION

This section evaluates MQ-GNN's performance across various GNN models and standard datasets. Specifically, we analyze its performance on both single-GPU and multi-GPU configurations, leveraging NS and LS sampling approaches.

1) SINGLE GPU PERFORMANCE

MQ-GNN demonstrates substantial performance gains in single-GPU configurations, achieving significantly faster training and batch processing times than baselines. MQ-GNN reduces training time by up to 2.06 × across four datasets while maintaining comparable evaluation metrics.

NS Performance NS methods such as MQ-GraphSAGE demonstrate significant performance improvements. For Reddit, with its dense graph structure and high-degree nodes, MQ-GCN achieved a 2.01 × speedup, as shown in Table 2, demonstrating the efficiency of MQ-GNN's queuing mechanism in managing data transfer and computation for large feature sizes and dense neighborhoods. Similarly, on the ogbn-products dataset, MQ-GraphSAGE reduced training time by 1.81 × and improved batch time by 1.98 ×. MQ-GNN achieves significant performance gains on the Reddit dataset due to its dense structure and large feature dimensions. The significant feature dimensions and dense connectivity enable the queuing mechanism to interleave data transfer and computation, improving GPU utilization efficiently. In contrast, the dense structure of ogbn-products facilitates the queuing mechanism, but its large graph size increases data transfer demands. Although the smaller feature size allows for some caching benefits, it is less impactful than Reddit, which has larger feature dimensions.

MQ-GCN maintains efficiency even on smaller graphs such as ogbn-proteins and sparse datasets such as ogbn-arxiv. Table 2 shows that MQ-GCN achieved a 1.5 × speedup on ogbn-arxiv by leveraging its moderate graph density and larger feature dimension (128). Due to the more prominent feature dimension, MQ-GNN's caching approach improves GPU efficiency by reducing redundant data transfers and effectively interleaving data transfer with GNN computation. In contrast, MQ-GCN reduced training time by 1.4 × on the ogbn-proteins dataset. The smaller feature dimension (8) limits the effectiveness of caching and reduces the efficiency of interleaving data transfer with GNN computation, resulting in comparatively more minor gains than on ogbn-arxiv.

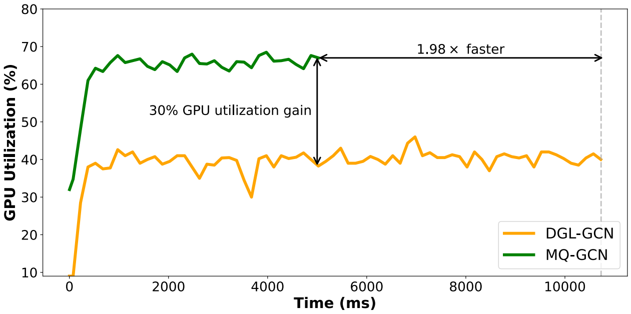

FIGURE 3. The GPU utilization of MQ-GCN and DGL-GCN during a single epoch of training on Reddit. Utilization is smoothed over 150 ms.

LS Performance LS methods benefited greatly from MQ-GNN's pipelining mechanism. For ogbn-products, MQ-FastGCN+f+d achieved a 2.06 × reduction in training time (Table 3), demonstrating its ability to manage the dataset's large scale and high feature dimensionality using multi-queue and caching mechanisms. For smaller graphs like ogbn-proteins, MQ-FastGCN+f+d reduced training time by 1.6 ×, while MQ-LADIES+f+d dropped it by 1.7 × (Tables 3 and 4), demonstrating MQ-GNN's versatility across varying dataset scales and graph structures. MQ-GNN's queuing method reduces GPU idle times and improves utilization by increasing the overlap between computation and data transfer. Mini-batch preparation is accelerated by the caching mechanism's ability to reduce redundant node access due to the dense graph connectivity.

Comparative Analysis Across all datasets and LS and NS strategies, MQ-GNN consistently outperformed baselines, achieving faster training times and batch processing speeds without compromising metrics. Separating sampling from training eliminates bottlenecks observed in traditional systems like LADIES, where sampling distribution updates (Section III-B) slow down mini-batch preparation. MQ-GNN's queuing mechanism overlaps data transfer, sampling, and computation phases to ensure continuous GPU utilization. Additionally, MQ-GNN effectively integrates processes with higher computational costs, such as debiasing (13) and flat sampling (10) in LADIES+f+d and FastGCN+f+d, into its pipeline to minimize its runtime impact.

We also compare GPU utilization during training on a single epoch of the Reddit dataset using MQ-GCN and GCN, as shown in Fig. 3, where MQ-GNN achieves a 1.98 × speedup and a 30% increase in GPU utilization. MQ-GNN achieves significantly higher GPU utilization, maintaining an average utilization of approximately 64.2% with a peak utilization of 73%, compared to DGL, which stabilizes around 39.42%. A key factor influencing GPU utilization is determining the optimal queue size and interleaving data transfer and GNN computation to reduce GPU starvation for data.

In MQ-GNN, the queue size directly affects the overlap between batch preparation and GPU computation. Our

TABLE 2. Results for batch time (ms), training time (ms), and evaluation metrics (%) for GCN, GraphSAGE, MQ-GCN, and MQ-GraphSAGE on a single-GPU configuration.

	Benchmark	ogbn-proteins	ogbn-arxiv	Reddit	ogbn-products
GCN	Batch	10.29 ± 0.20	11.53 ± 0.15	24.70 ± 0.18	14.11 ± 0.24
	Training	351669.12 ± 3616.96	505270.1 ± 4952.69	**1471507.1 ± 15604.08**	1081781.22 ± 21635.62
	Metrics	66.31 ± 0.50	67.69 ± 0.55	88.81 ± 0.80	76.10 ± 0.48
MQ-GCN	Batch	**6.43 ± 0.15**	**6.50 ± 0.20**	11.89 ± 0.18	6.83 ± 0.22
	Training	**197371.73 ± 1953.72**	**280705.61 ± 2807.06**	**731104.99 ± 8175.05**	559839.94 ± 6011.34
	Metrics	66.87 ± 0.35	67.25 ± 0.40	89.71 ± 0.55	76.95 ± 0.40
GraphSAGE	Batch	23.03 ± 0.25	28.11 ± 0.30	38.01 ± 0.35	21.23 ± 0.40
	Training	786946.56 ± 6993.38	1050783.26 ± 12418.31	2326703.17 ± 22372.31	**1725299.42 ± 25712.47**
	Metrics	66.01 ± 0.45	68.51 ± 0.55	94.36 ± 0.25	76.87 ± 0.65
MQ-GraphSAGE	Batch	15.99 ± 0.20	17.99 ± 0.25	23.36 ± 0.30	12.27 ± 0.25
	Training	536981.42 ± 4360.81	639797.48 ± 5837.68	1351612.87 ± 12926.13	**958494.12 ± 9584.94**
	Metrics	66.11 ± 0.35	68.41 ± 0.40	94.41 ± 0.30	77.01 ± 0.40

TABLE 3. Results for batch time (ms), training time (ms), and evaluation metrics (%) for FastGCN, MQ-FastGCN, and their enhanced variants on a single-GPU configuration.

Model	Benchmark	ogbn-proteins	ogbn-arxiv	Reddit	ogbn-products
FastGCN	Batch	8.99 ± 0.20	8.84 ± 0.25	9.75 ± 0.30	10.09 ± 0.50
	Training	307300.5 ± 3073.01	316598.2 ± 26256.19	588347.7 ± 4975.32	400561.65 ± 7831.35
	Metrics	53.2 ± 0.50	26.11 ± 0.40	45.7 ± 0.55	27.3 ± 0.55
MQ-FastGCN	Batch	5.42 ± 0.18	5.11 ± 0.20	5.39 ± 0.22	5.01 ± 0.25
	Training	180722.50 ± 1707.23	173888.99 ± 1758.89	298911.13 ± 3268.60	200101.12 ± 2225.34
	Metrics	53.1 ± 0.45	26.21 ± 0.40	45.9 ± 0.50	27.1 ± 0.50
FastGCN+f	Batch	8.57 ± 0.15	8.85 ± 0.18	9.18 ± 0.20	10.07 ± 0.45
	Training	292837.5 ± 2839.29	316811.5 ± 3057.12	553561.12 ± 5535.61	390916.17 ± 6929.41
	Metrics	61.1 ± 0.45	23.61 ± 0.35	53.61 ± 0.49	33.22 ± 0.51
MQ-FastGCN+f	Batch	5.36 ± 0.15	5.23 ± 0.18	5.74 ± 0.20	6.29 ± 0.22
	Training	162687.50 ± 1626.88	176006.39 ± 1760.06	307534.00 ± 3075.34	217175.65 ± 2171.76
	Metrics	61.12 ± 0.30	23.13 ± 0.25	54.11 ± 0.40	33.21 ± 0.35
FastGCN+d	Batch	8.52 ± 0.15	8.65 ± 0.18	9.31 ± 0.25	9.97 ± 0.50
	Training	291145.5 ± 2844.56	309764.5 ± 3284.55	561751.15 ± 4727.52	386924.19 ± 6736.37
	Metrics	53.69 ± 0.51	27.32 ± 0.65	44.25 ± 0.70	27.97 ± 0.65
MQ-FastGCN+d	Batch	5.23 ± 0.18	5.01 ± 0.20	5.12 ± 0.22	5.23 ± 0.25
	Training	181747.50 ± 1617.48	171091.39 ± 1720.91	310028.42 ± 3110.28	202957.88 ± 2149.58
	Metrics	53.75 ± 0.51	27.25 ± 0.65	44.66 ± 0.70	27.84 ± 0.65
FastGCN+f+d	Batch	8.65 ± 0.20	8.42 ± 0.18	9.47 ± 0.22	20.22 ± 0.60
	Training	**295582.1 ± 3023.79**	301607.1 ± 3161.71	571206.23 ± 4821.12	**784496.26 ± 15689.93**
	Metrics	61.1 ± 0.75	24.79 ± 0.70	53.97 ± 0.85	30.11 ± 1.50
MQ-FastGCN+f+d	Batch	5.39 ± 0.20	5.27 ± 0.22	5.94 ± 0.25	6.12 ± 0.30
	Training	**163400.35 ± 1634.00**	170154.30 ± 1701.54	297258.84 ± 2972.59	**378999.76 ± 3790.00**
	Metrics	61.13 ± 0.85	24.85 ± 0.70	53.99 ± 0.85	29.97 ± 1.50

analysis shows that datasets with higher sampling and data transfer times, such as Reddit, required larger queue sizes to utilize the GPU fully. Conversely, smaller datasets like ogbn-arxiv, with lower variability in batch preparation times, achieved better performance with smaller queue sizes.

MQ-GNN dynamically adjusted queue sizes to maximize throughput while avoiding memory overflows by profiling workload characteristics and applying a formula (24) that accounts for maximum preparation time, average compute time and memory constraints. For example, in the Reddit dataset, where the peak memory usage was 730.95 MB and the maximum combined sampling and data transfer time was 51 ms, a queue size of 5 ensured steady GPU utilization without exceeding memory limits. Similarly, in the ogbn-products dataset, where peak memory usage was 240.62 MB and preparation times were lower, a smaller queue size of 3 was sufficient to maintain optimized GPU utilization.

These results highlight the adaptability of MQ-GNN in addressing dataset-specific challenges and ensuring efficient memory usage and consistent GPU utilization. MQ-GNN demonstrated higher peak GPU utilization and more excellent stability than DGL, which exhibited frequent dips in utilization, likely caused by sequential data processing and transfer operations that introduced delays. This stability results from integrating queuing mechanisms and memory-based constraints that optimize data pipelines.

In addition, MQ-GNN prevents GPU memory saturation for large datasets by limiting queue sizes according to available GPU memory and batch size requirements. MQ-GNN's profile-driven queue sizing achieves better utilization and balances resource constraints and computational efficacy. This method involves profiling peak memory usage per dataset and determining optimal queue sizes under GPU memory limits, considering temporary data utilized while performing CUDA operations. As a result, this approach

TABLE 4. Results for batch time (ms), training time (ms), and evaluation metrics (%) for LADIES, MQ-LADIES, and their enhanced variants on a single-GPU configuration.

Model	Benchmark	ogbn-proteins	ogbn-arxiv	Reddit	ogbn-products
LADIES	Batch	10.18 ± 0.20	11.39 ± 0.15	13.26 ± 0.18	9.11 ± 0.40
	Training	348144.5 ± 4021.12	**604201.54 ± 5898.82**	807788.34 ± 7965.98	476977.05 ± 9539.54
	Metrics	68.27 ± 0.12	61.0 ± 0.20	74.9 ± 0.18	52.84 ± 0.40
MQ-LADIES	Batch	6.51 ± 0.18	7.11 ± 0.20	7.64 ± 0.18	4.99 ± 0.25
	Training	211709.12 ± 2427.09	**348106.70 ± 2851.07**	442648.37 ± 5726.48	248904.21 ± 2359.04
	Metrics	68.37 ± 0.10	61.4 ± 0.15	75.0 ± 0.20	52.71 ± 0.25
LADIES+f	Batch	10.24 ± 0.22	11.27 ± 0.18	13.29 ± 0.20	9.13 ± 0.50
	Training	349983.5 ± 2799.98	493595.2 ± 5365.12	805818.11 ± 7933.98	528674.09 ± 10573.48
	Metrics	68.18 ± 0.10	62.8 ± 0.18	90.12 ± 0.20	62.14 ± 0.30
MQ-LADIES+f	Batch	6.35 ± 0.18	6.61 ± 0.22	7.36 ± 0.18	4.79 ± 0.25
	Training	201255.32 ± 2432.55	269853.61 ± 2858.54	416927.39 ± 5769.27	247821.93 ± 2378.22
	Metrics	68.38 ± 0.15	62.82 ± 0.20	90.21 ± 0.25	62.11 ± 0.40
LADIES+d	Batch	11.62 ± 0.25	13.0 ± 0.22	14.28 ± 0.25	9.18 ± 0.60
	Training	397178.5 ± 2989.76	571631.9 ± 4979.89	1063068.25 ± 11520.23	432755.37 ± 8655.11
	Metrics	68.76 ± 0.15	61.48 ± 0.22	86.81 ± 0.25	54.82 ± 0.50
MQ-LADIES+d	Batch	7.31 ± 0.25	7.98 ± 0.22	8.40 ± 0.25	5.09 ± 0.30
	Training	248188.67 ± 2521.89	334029.90 ± 2990.30	590965.54 ± 5929.66	239021.27 ± 2510.21
	Metrics	68.9 ± 0.20	61.45 ± 0.25	86.89 ± 0.30	54.98 ± 0.35
LADIES+f+d	Batch	11.58 ± 0.22	**14.96 ± 0.25**	15.35 ± 0.30	9.11 ± 0.70
	Training	395726.1 ± 2989.67	697795.4 ± 5969.87	1381972.57 ± 14728.65	404781.39 ± 15895.63
	Metrics	67.55 ± 0.20	61.90 ± 0.25	88.43 ± 0.30	62.20 ± 0.60
MQ-LADIES+f+d	Batch	7.09 ± 0.22	7.55 ± 0.25	8.45 ± 0.30	5.01 ± 0.30
	Training	231872.88 ± 2518.73	383751.93 ± 2997.52	694872.38 ± 5948.72	201580.35 ± 2515.80
	Metrics	67.49 ± 0.25	62.01 ± 0.30	88.29 ± 0.35	62.30 ± 0.40

enhances GPU utilization. These results demonstrate the advantages of MQ-GNN's queuing mechanism and asynchronous data handling, which sustain GPU activity and enhance overall throughput.

MQ-GNN's lightweight design eliminates the gradient queue for single-GPU configurations, simplifies the training pipeline, and reduces training time. Its queuing and pipelining mechanisms enable these improvements by decoupling sampling and computation, increasing the overlap between data transfer and training. By avoiding sequential processing, where GPUs are often idle during data preparation, MQ-GNN eliminates these bottlenecks, resulting in consistent GPU utilization and improved throughput. These findings establish MQ-GNN as a robust, scalable, and efficient solution for GNN training, even in single-GPU setups.

2) MULTI-GPU PERFORMANCE

MQ-GNN provides significant performance gains in multi-GPU configurations compared to baselines (up to 4.6 ×), as evident from Tables 5-13. By leveraging the RaCoM framework (Section IV), gradient sharing with queues, and efficient queuing mechanisms, MQ-GNN demonstrates faster training and batch times across diverse datasets while maintaining comparable evaluation performance despite a drop of approximately 2.0% in evaluation metrics. These results highlight the system's ability to reduce communication overhead, minimize GPU idle time, and handle staleness issues through periodic synchronization.

NS Performance In a two-GPU configuration, MQ-GNN achieves substantial reductions in training and batch times while maintaining competitive metrics, as shown in Tables 5-7. For example, MQ-GCN significantly improved over GCN on the Reddit dataset, with a training time reduction of 3.10 ×. MQ-GraphSAGE also achieved a 3.15 × speedup (Table 5). These results highlight MQ-GNN's efficient queuing mechanism, which reduces GPU idle time and enables smooth mini-batch transitions. Despite these improvements, there was a minor decline in metrics, with MQ-GCN experiencing a decline of 0.05% and MQ-GraphSAGE a decline of 0.2%. The staleness caused by asynchronous updates during gradient sharing and model updates causes this decline. RaCoM does not eliminate the effects of staleness, especially in models that are more susceptible to asynchronous updates. However, it reduces significantly through periodic synchronization and appropriate gradient management.

For smaller datasets like ogbn-proteins, MQ-GraphSAGE provides 2.61 × improvement. Batch time decreases proportionally, emphasizing MQ-GNN's ability to adapt to varying graph sizes. Because of the sparsity of the graphs and infrequent model synchronization, MQ-GNN exhibits a more considerable decrease in metrics for smaller datasets like ogbn-proteins and ogbn-arxiv. On ogbn-proteins, MQ-GraphSAGE's evaluation metric decreased by 0.3%, whereas on ogbn-arxiv, it decreased by 0.28% (Table 5). The sparse connectivity of these datasets reduces the benefits of caching and queuing, and infrequent synchronization increases the impact of staleness in asynchronous updates, degrading overall model accuracy.

Scaling up to three and four GPUs, MQ-GNN continues to outperform baselines in training and batch times, as evidenced by Tables 8 to 13 in Appendix B. For the Reddit dataset, MQ-GCN delivers a remarkable 3.95 × improvement on three GPUs, as shown in Table 8. Similarly, for the

TABLE 5. Results for batch time (ms), training time (ms), and evaluation metrics (%) for GCN, GraphSAGE, MQ-GCN, and MQ-GraphSAGE on a two-GPU configuration.

Model	Benchmark	ogbn-proteins	ogbn-arxiv	Reddit	ogbn-products
GCN	Batch	12.15 ± 0.58	13.84 ± 0.47	**23.54 ± 0.42**	13.32 ± 0.49
	Training	207644.09 ± 11074.32	298242.34 ± 3345.21	**716904.12 ± 7458.76**	224901.25 ± 5289.03
	Metrics	66.28 ± 0.49	67.71 ± 0.52	**88.35 ± 0.79**	75.36 ± 0.51
MQ-GCN	Batch	4.81 ± 8.25	5.20 ± 9.12	**7.47 ± 22.50**	4.61 ± 10.34
	Training	83392.37 ± 0.49	114318.65 ± 0.52	**230592.43 ± 0.79**	80000.44 ± 0.51
	Metrics	66.08 ± 0.40	67.61 ± 0.45	**88.3 ± 0.70**	75.26 ± 0.40
GraphSAGE	Batch	23.71 ± 0.40	30.85 ± 0.61	36.13 ± 0.55	23.09 ± 0.37
	Training	**405154.41 ± 4358.29**	637116.36 ± 6728.41	**1132782.14 ± 11834.12**	504154.02 ± 5310.89
	Metrics	**66.03 ± 0.42**	**68.58 ± 0.57**	94.89 ± 0.23	**78.39 ± 0.62**
MQ-GraphSAGE	Batch	8.91 ± 12.51	11.41 ± 15.34	11.21 ± 27.56	7.62 ± 13.28
	Training	**154708.64 ± 0.50**	241777.02 ± 0.52	**359521.25 ± 0.80**	171465.83 ± 0.62
	Metrics	**65.73 ± 0.35**	**68.38 ± 0.40**	94.84 ± 0.18	**78.19 ± 0.50**

ogbn-products dataset, MQ-GraphSAGE achieved a 3.36 × speedup. For the ogbn-arxiv dataset, MQ-GraphSAGE achieves a 2.89 × speedup in training time. The batch time decreases significantly, from 31.59 ms to 10.45 ms (Table 8). With four GPUs, MQ-GNN maintained its efficiency, as seen on the ogbn-arxiv dataset, where MQ-GCN achieved a 3.45 × improvement in training time (Table 11). On dense datasets such as Reddit and ogbn-products, MQ-GNN excels because it leverages caching and overlapping for optimal efficiency. MQ-GNN's architecture efficiently lowers synchronization and communication overhead, especially in dense graphs. These findings show that MQ-GNN can efficiently manage dense datasets, even though synchronization becomes more difficult as the number of GPUs increases.

LS Performance MQ-GNN achieved significant speed improvements over layer-wise approaches for two GPUs, as shown in Tables 6 and 7. On the Reddit dataset using MQ-FastGCN, the training time improves by 3.26 × (Table 6). Similarly, MQ-LADIES provides a 3.23 × reduction in training time (Table 7). On the ogbn-products dataset, MQ-FastGCN+f+d achieved a 2.92 × improvement, with batch times decreasing by 2.97 × and a slight 0.2% drop in the metric. For dense graphs like Reddit, MQ-LADIES+f experiences a smaller decrease in its evaluation metric, with a 0.05% drop compared to its counterpart, while achieving a 3.27 × speedup. These results emphasize MQ-GNN's ability to integrate additional computational overhead, such as flat sampling (10) and debiasing (13), in advanced variants of LADIES and FastGCN, with minimal runtime overhead.

As the number of GPUs increases to three and four, MQ-GNN continues to show remarkable efficiency, as shown in Tables 8-13 in Appendix B. On three GPUs, MQ-FastGCN achieves a 3.38 × improvement, as presented in Table 9. Similarly, on the ogbn-arxiv dataset, MQ-LADIES reduces training time by 2.95 ×. Batch time follows a similar trend, dropping from 10.92 ms to 3.52 ms (Table 10). Despite the training speedup, evaluation metrics experience a slightly larger drop due to staleness, with MQ-FastGCN+f+d dropping the metric by 0.8% (Table 9). MQ-LADIES+f+d demonstrates strong scalability on four GPUs, achieving a 3.6 × reduction in training time for the ogbn-products dataset. Batch time improves proportionally, dropping from 8.36 ms to 2.23 ms at the cost of a 1.0% reduction in the metric, showcasing MQ-GNN's ability to dynamically balance staleness and synchronization costs, as presented in Table 13. On the Reddit dataset, MQ-LADIES+f+d and MQ-FastGCN+f+d provide a 4.35 × speedup in training, with a slight 0.6% drop in the metric (Tables 13 and 12). Even with three and four GPUs, MQ-GNN maintains evaluation metrics close to its baselines, with only minor deviations due to reduced staleness through periodic synchronization using RaCoM (Section IV). Furthermore, the integration of additional computational overhead, such as flat sampling (10) and debiasing (13) in LADIES+f+d and FastGCN+f+d, is efficiently managed within MQ-GNN's pipeline without significantly increasing runtime. This is achieved by efficiently overlapping these tasks with other operations. Certain models, such as LADIES, are more sensitive to staleness caused by asynchronous updates, which affect evaluation metrics, particularly in scenarios with infrequent synchronization.

Comparative Analysis By aligning updates at regular intervals, RaCoM ensures that the model remains synchronized, even in scenarios with high communication demands. This approach allows MQ-GNN to deliver stable evaluation metrics across configurations with two or three GPUs, with slight declines when scaling to four GPUs, particularly on smaller or sparse datasets, such as ogbn-proteins.

Graph sparsity amplifies the effects of staleness because the periodic synchronization intervals in RaCoM cannot fully address the asynchronous updates required for consistent feature aggregation. For instance, in (1), staleness can disrupt the aggregation process when it uses outdated node embeddings (Z^{l-1}) due to delayed gradient updates. This leads to inconsistent feature propagation, which is especially problematic in sparse datasets, where every edge and feature has a more pronounced impact on the overall aggregation. Similarly, in GraphSAGE, staleness disrupts the sampling process when neighbor embeddings (3) are not synchronized. Sparse graphs worsen this issue because of a limited number

TABLE 6. Results for batch time (ms), training time (ms), and evaluation metrics (%) for FastGCN, MQ-FastGCN, and their enhanced variants on two-GPU configuration.

Model	Benchmark	ogbn-proteins	ogbn-arxiv	Reddit	ogbn-products
FastGCN	Batch	8.04 ± 0.37	7.46 ± 0.28	9.08 ± 0.41	10.36 ± 0.45
	Training	137406.91 ± 1421.17	133517.13 ± 1402.34	**273813.32 ± 2845.98**	151858.83 ± 1559.87
	Metrics	53.21 ± 0.48	26.19 ± 0.39	45.81 ± 0.55	28.88 ± 0.52
MQ-FastGCN	Batch	2.90 ± 8.21	2.61 ± 7.89	2.72 ± 10.45	3.15 ± 9.34
	Training	51189.87 ± 0.55	48299.73 ± 0.40	**83994.41 ± 0.60**	47605.60 ± 0.49
	Metrics	52.71 ± 0.40	25.89 ± 0.35	45.71 ± 0.48	28.58 ± 0.45
FastGCN+f	Batch	8.04 ± 0.42	7.87 ± 0.39	8.93 ± 0.48	8.78 ± 0.43
	Training	137441.27 ± 1457.12	140884.09 ± 1482.94	269335.06 ± 2798.42	128694.32 ± 1351.89
	Metrics	61.12 ± 0.48	26.20 ± 0.65	53.64 ± 0.69	33.24 ± 0.63
MQ-FastGCN+f	Batch	3.04 ± 8.32	2.91 ± 8.45	2.75 ± 10.75	2.98 ± 9.15
	Training	52834.47 ± 0.52	53127.50 ± 0.45	84174.25 ± 0.63	44628.97 ± 0.58
	Metrics	60.72 ± 0.40	26.0 ± 0.50	53.54 ± 0.60	33.04 ± 0.50
FastGCN+d	Batch	8.05 ± 0.39	8.06 ± 0.35	8.79 ± 0.47	9.59 ± 0.52
	Training	137557.27 ± 1435.89	144348.22 ± 1487.32	265117.67 ± 2759.83	140487.23 ± 1453.76
	Metrics	53.70 ± 0.72	27.37 ± 0.68	44.28 ± 0.83	28.01 ± 1.43
MQ-FastGCN+d	Batch	3.12 ± 8.57	3.07 ± 8.94	2.75 ± 11.10	3.35 ± 9.78
	Training	54547.94 ± 0.62	55924.65 ± 0.65	84382.63 ± 0.73	50125.19 ± 0.85
	Metrics	53.20 ± 0.60	27.07 ± 0.60	44.18 ± 0.70	27.71 ± 1.20
FastGCN+f+d	Batch	7.67 ± 0.43	7.66 ± 0.35	8.88 ± 0.50	9.59 ± 0.34
	Training	131084.62 ± 1348.91	137053.24 ± 1421.38	267829.32 ± 2823.94	**140487.28 ± 1459.75**
	Metrics	61.15 ± 0.12	24.32 ± 0.18	53.90 ± 0.19	**30.14 ± 0.37**
MQ-FastGCN+f+d	Batch	2.82 ± 8.11	2.77 ± 8.45	2.69 ± 10.98	3.22 ± 9.05
	Training	49439.57 ± 0.50	50670.86 ± 0.45	82618.40 ± 0.58	**48081.08 ± 0.55**
	Metrics	60.75 ± 0.10	24.12 ± 0.15	53.80 ± 0.12	**29.94 ± 0.35**

TABLE 7. Results for batch time (ms), training time (ms), and evaluation metrics (%) for LADIES, MQ-LADIES, and their enhanced variants on two-GPU configuration.

	Benchmark	ogbn-proteins	ogbn-arxiv	Reddit	ogbn-products
LADIES	Batch	6.85 ± 0.19	10.92 ± 0.41	8.20 ± 0.33	7.59 ± 0.28
	Training	117160.16 ± 1198.45	235377.73 ± 2430.52	**191979.81 ± 1998.45**	366542.86 ± 3721.24
	Metrics	68.34 ± 0.16	60.62 ± 0.23	75.47 ± 0.22	53.47 ± 0.42
MQ-LADIES	Batch	2.49 ± 5.25	3.90 ± 8.90	2.49 ± 11.25	2.41 ± 9.95
	Training	43320.33 ± 0.08	85432.90 ± 0.11	**59420.67 ± 0.13**	118779.52 ± 0.15
	Metrics	67.94 ± 0.10	60.42 ± 0.20	75.42 ± 0.18	53.27 ± 0.40
LADIES+f	Batch	6.72 ± 0.24	10.60 ± 0.43	8.43 ± 0.48	8.23 ± 0.36
	Training	114955.31 ± 1203.42	222753.14 ± 2299.01	**198578.13 ± 2011.68**	{438722.13} ± 4487.24
	Metrics	68.66 ± 0.07	63.45 ± 0.14	**90.55 ± 0.16**	**63.21 ± 0.22**
MQ-LADIES+f	Batch	2.44 ± 5.15	3.75 ± 8.65	2.53 ± 10.95	2.53 ± 10.25
	Training	42378.54 ± 0.09	80344.90 ± 0.13	**60673.42 ± 0.15**	**137522.97 ± 0.22**
	Metrics	68.36 ± 0.08	63.25 ± 0.15	**90.50 ± 0.12**	**63.01 ± 0.20**
LADIES+d	Batch	11.05 ± 0.53	13.68 ± 0.39	9.84 ± 0.47	7.85 ± 0.34
	Training	188865.71 ± 1924.32	327562.53 ± 3321.40	443242.73 ± 4520.24	451953.42 ± 4615.22
	Metrics	68.77 ± 0.11	61.43 ± 0.17	87.71 ± 0.18	55.58 ± 0.37
MQ-LADIES+d	Batch	4.03 ± 8.35	5.01 ± 10.78	3.08 ± 12.45	2.53 ± 11.35
	Training	70227.20 ± 0.12	122060.70 ± 0.14	141574.51 ± 0.18	147443.91 ± 0.23
	Metrics	68.37 ± 0.12	61.13 ± 0.18	87.61 ± 0.18	55.28 ± 0.35
LADIES+f+d	Batch	11.78 ± 0.41	12.80 ± 0.34	10.69 ± 0.50	7.80 ± 0.42
	Training	201339.29 ± 2034.67	280377.04 ± 2863.15	481194.71 ± 4895.74	452177.30 ± 4610.84
	Metrics	67.56 ± 0.16	62.99 ± 0.18	88.94 ± 0.24	63.53 ± 0.48
LADIES+f+d	Batch	4.52 ± 8.95	4.77 ± 10.95	3.23 ± 13.10	2.67 ± 12.15
	Training	78953.76 ± 0.11	106808.64 ± 0.15	147969.36 ± 0.20	158074.05 ± 0.28
	Metrics	67.16 ± 0.15	62.79 ± 0.18	88.84 ± 0.20	63.33 ± 0.45

of neighbors, making embedding of each neighbor critical. Thus, RaCoM's synchronization periods do not align well with updates, resulting in inconsistent aggregation and degraded performance.

Furthermore, staleness in the RaCoM model also affects LS approaches. For instance, FastGCN, as defined in (4), samples nodes at each layer, assuming independence among sampled nodes. Thus, delayed synchronization impacts FastGCN because it struggles with outdated sampled embeddings that fail to represent the current state of the graph, leading to estimation errors and high variance in the aggregation process. Dense datasets partially mitigate this issue due to redundancy, but sparse graphs amplify the problem as fewer nodes are sampled. Compared to FastGCN, LADIES is more severely affected because each layer depends on sampled nodes from the previous layer, as defined in (6). The hierarchical dependency between layers further magnifies the effects of staleness. When embedding updates from

TABLE 8. Results for batch time (ms), training time (ms), and evaluation metrics (%) for GCN, GraphSAGE, MQ-GCN, and MQ-GraphSAGE on a three-GPU configuration.

	Benchmark	ogbn-proteins	ogbn-arxiv	Reddit	ogbn-products
GCN	Batch	13.58 ± 0.12	13.85 ± 0.10	24.25 ± 0.18	15.19 ± 0.11
	Training	155635.53 ± 1525.12	249346.67 ± 2050.21	**506732.17** ± **3810.48**	296226.39 ± 1820.34
	Metrics	66.79 ± 0.70	68.45 ± 0.50	90.82 ± 0.35	71.04 ± 0.42
MQ-GCN	Batch	4.52 ± 0.18	4.39 ± 0.25	5.91 ± 0.30	4.22 ± 0.20
	Training	54487.44 ± 2400.35	**83083.30** ± **3260.28**	**128234.18** ± **5210.45**	**85834.73** ± **2100.48**
	Metrics	66.29 ± 0.35	68.15 ± 0.42	90.62 ± 0.60	70.74 ± 0.50
GraphSAGE	Batch	25.01 ± 0.64	**31.59** ± **0.43**	38.29 ± 0.26	23.65 ± 0.59
	Training	277099.73 ± 0.16	**432195.54** ± **0.28**	777316.19 ± 0.94	**378204.24** ± **0.05**
	Metrics	66.62 ± 0.98	69.07 ± 0.47	94.53 ± 0.95	63.72± 0.26
MQ-GraphSAGE	Batch	8.50 ± 4.12	**10.45** ± **5.85**	9.45 ± 7.15	6.76 ± 5.05
	Training	98610.54 ± 110.25	**149490.10** ± **150.35**	199308.66 ± 220.18	**112520.80** ± **175.45**
	Metrics	65.92 ± 0.32	67.66 ± 0.38	90.83 ± 0.15	70.74 ± 0.45

earlier layers are asynchronous, the approximation quality in subsequent layers suffers, particularly in sparse graphs. MQ-GNN's periodic synchronization through RaCoM does not align effectively with the layer-dependent sampling of LADIES, resulting in higher estimation errors and degraded performance. While flat sampling (10) and debiasing (13) techniques for FastGCN and LADIES aim to improve sampling robustness by reducing variance and correcting estimation biases, their effectiveness diminishes in multi-GPU environments where delays in gradient synchronization introduce non-uniformity in sampling distributions. This introduces another layer of complexity that is difficult for MQ-GNN's current synchronization strategy to address completely.

In multi-GPU systems, the pipelining method with multi-queuing in MQ-GNN is essential for maximizing resource efficiency. The method enqueues a predetermined number of mini-batches to reduce GPU idle time and guarantee seamless task transitions. The queuing strategy improves training efficiency and performs well across various datasets and configurations. This method works well for large datasets, where scalability depends on sustaining high throughput across several GPUs.

Integrating these techniques—RaCoM, multi-queue, and GNS—makes MQ-GNN a robust framework for multi-GPU setups. The periodic synchronization provided by RaCoM mitigates the impact of asynchronous processing and communication delays. The multi-queue approach enhances GPU utilization by decreasing idle time and guaranteeing seamless training. By reducing communication overhead in data transfer to GPU, GNS further accelerates the training. Together, these techniques allow MQ-GNN to achieve scalable and efficient performance, albeit with a slight drop in metrics across large and small datasets. These improvements show MQ-GNN's effectiveness as a scalable alternative to existing baselines in multi-GPU environments. While MQ-GNN provides significant computational speedups and scalability, its synchronization and queuing mechanisms are less effective at stabilizing the performance of layer-wise sampling models under multi-GPU training.

VI. CONCLUSION

This paper presented MQ-GNN, a scalable and efficient framework for multi-GPU GNN training. MQ-GNN employs multi-queue pipelining to efficiently overlap mini-batch generation, data transfer, and computation, reducing idle times and maximizing resource utilization. It incorporates an adaptive queue-sizing strategy to balance memory constraints and computational efficiency while leveraging global neighbor sampling with caching to minimize data transfer overhead. Furthermore, the RaCoM periodic synchronization minimizes communication bottlenecks while maintaining model accuracy considering the graph characteristics and number of GPUs.

Experimental results on four large-scale datasets and ten baseline models demonstrate that MQ-GNN delivers up to 4.6 × faster training and 30% higher GPU utilization, with only a 2% accuracy trade-off. These results validate MQ-GNN's effectiveness in optimizing large-scale GNN training across diverse graph structures.

MQ-GNN offers a high-performance and scalable graph learning solution for applications that include recommendation engines, social networks, and biological systems. Future research will extend MQ-GNN to dynamic graphs and more complex GNN structures to further improve its suitability for changing large-scale datasets.

APPENDIX A
PERIODIC SYNCHRONIZATION

Training GNNs involves three primary factors: node-wise computations, edge-wise computations, and GPU workload. Node-wise computations scale with $|V|$, contributing to the workload through node embeddings and local feature transformations. Larger $|V|$ requires a longer synchronization interval to balance computational cost while maintaining efficiency. Edge-wise computations scale with $|E|$, contributing to neighbor aggregation and communication overhead. The square root of $|E|$ models the diminishing returns of additional edges on staleness, ensuring that denser graphs synchronize more frequently to prevent divergence caused by asynchronous updates. GPU workload distribution further

TABLE 9. Results for batch time (ms), training time (ms), and evaluation metrics (%) for FastGCN, MQ-FastGCN, and their enhanced variants on three-GPU configuration.

	Benchmark	ogbn-proteins	ogbn-arxiv	Reddit	ogbn-products
FastGCN	Batch	7.82 ± 0.35	8.09 ± 0.30	9.21 ± 0.40	9.62 ± 0.35
	Training	89664.64 ± 725	97615.04 ± 810	185078.34 ± 1290	**94393.48 ± 520**
	Metrics	53.69 ± 0.35	26.9 ± 0.42	43.61 ± 0.50	28.81 ± 0.45
MQ-FastGCN	Batch	2.74 ± 1.50	2.70 ± 1.80	2.30 ± 3.25	2.73 ± 2.40
	Training	33072.93 ± 125.35	34224.56 ± 135.45	48058.41 ± 155.25	**27917.82 ± 145.40**
	Metrics	52.69 ± 0.20	26.1 ± 0.25	43.31 ± 0.25	28.11 ± 0.40
FastGCN+f	Batch	7.69 ± 0.20	7.81 ± 0.15	8.54 ± 0.25	9.02 ± 0.18
	Training	88148.54 ± 680	70788.87 ± 590	173517.37 ± 1200	89131.48 ± 610
	Metrics	62.1 ± 0.40	22.46 ± 0.35	52.99 ± 0.40	32.29 ± 0.32
MQ-FastGCN+f	Batch	2.66 ± 1.70	2.56 ± 2.00	2.08 ± 3.18	2.52 ± 2.20
	Training	32371.82 ± 130.25	24400.47 ± 110.25	43924.31 ± 145.30	26060.72 ± 125.45
	Metrics	61.3 ± 0.22	21.86 ± 0.25	52.69 ± 0.22	31.69 ± 0.42
FastGCN+d	Batch	7.62 ± 0.06	7.70 ± 0.37	8.75 ± 0.87	9.54 ± 1.05
	Training	87327.23 ± 0.96	69783.84 ± 0.62	175894.47 ± 0.92	93658.32 ± 0.05
	Metrics	54.22 ± 1.04	27.0 ± 0.76	44.93 ± 0.29	30.09 ± 0.38
MQ-FastGCN+d	Batch	2.71 ± 1.75	2.64 ± 1.90	2.24 ± 3.45	2.80 ± 2.75
	Training	32623.50 ± 125.45	25098.34 ± 130.25	46716.02 ± 145.80	28718.33 ± 135.45
	Metrics	53.32 ± 0.25	26.30 ± 0.30	44.53 ± 0.20	29.29 ± 0.40
FastGCN+f+d	Batch	7.85 ± 0.22	8.01 ± 0.74	8.91± 0.7	9.54± 0.95
	Training	**89942.67 ± 1.01**	96594.43 ± 0.01	179902.72 ± 0.38	93632.14 ± 0.64
	Metrics	**61.76 ± 0.07**	24.25 ± 0.2	54.53 ± 0.96	32.23 ± 0.88
MQ-FastGCN+f+d	Batch	2.77 ± 1.80	2.67 ± 2.15	2.20 ± 3.30	2.72 ± 2.50
	Training	**33275.04 ± 135.40**	33642.75 ± 140.20	45964.57 ± 150.30	27860.07 ± 130.35
	Metrics	**60.96 ± 0.15**	23.75 ± 0.20	54.23 ± 0.20	31.73 ± 0.35

TABLE 10. Results for batch time (ms), training time (ms), and evaluation metrics (%) for LADIES, MQ-LADIES, and their enhanced variants on three-GPU configuration.

	Benchmark	ogbn-proteins	ogbn-arxiv	Reddit	ogbn-products
LADIES	Batch	6.85 ± 0.20	10.92 ± 0.35	8.20 ± 0.30	7.59 ± 0.25
	Training	117160.16 ± 910.25	**235377.73 ± 1850.52**	191979.81 ± 1510.34	366542.86 ± 2760.28
	Metrics	68.34 ± 0.18	60.62 ± 0.28	75.47 ± 0.16	53.47 ± 0.30
MQ-LADIES	Batch	2.38 ± 0.54	3.52 ± 0.58	1.97 ± 0.16	2.08 ± 0.47
	Training	42568.80 ± 980.25	**79730.09 ± 1785.50**	47994.18 ± 1687.45	104707.20 ± 1520.37
	Metrics	67.44 ± 0.15	60.02 ± 0.33	75.17 ± 0.37	52.97 ± 0.74
LADIES+f	Batch	6.02 ± 0.20	10.30 ± 0.30	8.51 ± 0.35	8.12 ± 0.15
	Training	68952.23 ± 520.10	134775.56 ± 950.45	209233.72 ± 1320.75	150464.29 ± 920.30
	Metrics	68.67 ± 0.12	63.30 ± 0.20	90.87 ± 0.25	62.41 ± 0.18
MQ-LADIES+f	Batch	2.06 ± 0.42	3.26 ± 0.38	2.02 ± 0.18	2.19 ± 0.22
	Training	24675.93 ± 920.45	44787.82 ± 1187.35	51639.00 ± 1520.62	42384.17 ± 1387.44
	Metrics	67.87 ± 0.13	62.80 ± 0.23	90.67 ± 0.15	62.01 ± 0.30
LADIES+d	Batch	10.87 ± 0.35	12.34 ± 0.40	9.97 ± 0.28	8.02 ± 0.20
	Training	124582.76 ± 980.15	222182.65 ± 1450.52	286160.53 ± 1980.85	137120.38 ± 1150.30
	Metrics	69.30 ± 0.15	62.29 ± 0.18	88.43 ± 0.14	54.91 ± 0.24
MQ-LADIES+d	Batch	3.80 ± 0.38	4.18 ± 0.28	2.49 ± 0.45	2.30 ± 0.22
	Training	45733.19 ± 1250.45	78674.80 ± 1487.34	74325.03 ± 2287.64	41063.36 ± 1350.52
	Metrics	68.40 ± 0.15	61.69 ± 0.28	88.13 ± 0.26	54.31 ± 0.42
LADIES+f+d	Batch	11.82 ± 0.40	12.88 ± 0.35	9.97 ± 0.28	8.01 ± 0.20
	Training	135476.01 ± 1080.34	231789.48 ± 1630.25	229440.32 ± 1970.56	131150.32 ± 1140.78
	Metrics	68.06 ± 0.12	60.01 ± 0.18	88.78 ± 0.15	62.11 ± 0.22
MQ-LADIES+f+d	Batch	4.22 ± 0.28	4.43 ± 0.38	2.52 ± 0.12	2.35 ± 0.28
	Training	50542.12 ± 1025.35	83561.08 ± 1820.44	60232.61 ± 2350.54	40353.69 ± 1425.87
	Metrics	67.26 ± 0.23	59.51 ± 0.72	88.48 ± 0.25	61.61 ± 0.35

influences synchronization; as $|\mathcal{G}|$ increases, the workload is spread across more devices, shortening the synchronization interval to enhance consistency across distributed systems.

Proposition 1: The optimal periodic synchronization interval P_{ms}, which balances computational efficiency and model consistency, is given by:

$$P_{ms} = \left\lceil \frac{\sqrt{|V|}}{\sqrt{|\mathcal{G}|\,|E|}} \right\rceil \tag{26}$$

This interval ensures:

- *The synchronization frequency dynamically adapts to the graph density.*
- *The trade-off between staleness (due to asynchronous updates) and synchronization overhead (caused by frequent communication) is minimized.*

Proof: Let $C(P)$ be the total cost function that balances staleness and synchronization costs:

$$C(P) = \alpha P|E| + \beta \frac{1}{P}\frac{|V|}{|\mathcal{G}|},$$

TABLE 11. Results for batch time (ms), training time (ms), and evaluation metrics (%) for GCN, GraphSAGE, MQ-GCN, and MQ-GraphSAGE on a four-GPU configuration.

	Benchmark	ogbn-proteins	ogbn-arxiv	Reddit	ogbn-products
GCN	Batch	13.17 ± 0.15	14.48 ± 0.09	22.42 ± 0.85	15.70 ± 0.38
	Training	113834.39 ± 845.32	**173942.63 ± 1378.41**	351391.48 ± 2825.67	226895.51 ± 1545.27
	Metrics	66.81 ± 0.80	68.54 ± 0.40	90.97 ± 0.70	71.40 ± 0.45
MQ-GCN	Batch	3.87 ± 0.25	4.02 ± 0.30	4.87 ± 0.50	3.92 ± 0.45
	Training	34954.27 ± 265.15	**50400.06 ± 290.27**	78948.03 ± 400.67	58925.16 ± 335.28
	Metrics	65.61 ± 0.40	67.74 ± 0.45	90.47 ± 0.55	70.80 ± 0.50
GraphSAGE	Batch	20.54 ± 0.32	32.08 ± 0.25	39.36 ± 0.60	24.38 ± 0.50
	Training	177562.71 ± 1260.32	353692.29 ± 2058.45	**602149.33 ± 3450.87**	298715.49 ± 1885.40
	Metrics	66.62 ± 0.65	69.33 ± 0.50	**94.79 ± 0.30**	63.85 ± 0.40
MQ-GraphSAGE	Batch	6.14 ± 0.50	9.14 ± 0.60	8.65 ± 0.70	6.25 ± 0.60
	Training	55442.75 ± 385.45	105458.24 ± 500.62	**136716.51 ± 670.54**	79655.60 ± 450.27
	Metrics	65.12 ± 0.45	68.33 ± 0.50	**94.19 ± 0.35**	62.95 ± 0.42

TABLE 12. Results for batch time (ms), training time (ms), and evaluation metrics (%) for FastGCN, MQ-FastGCN, and their enhanced variants on four-GPU configuration.

	Benchmark	ogbn-proteins	ogbn-arxiv	Reddit	ogbn-products
FastGCN	Batch	7.85 ± 0.97	7.84 ± 0.85	9.24 ± 0.75	12.75 ± 0.58
	Training	67839.25 ± 525.45	53289.46 ± 395.25	139282.58 ± 865.35	125574.71 ± 710.15
	Metrics	53.69 ± 0.50	26.90 ± 0.55	43.67 ± 0.35	27.01 ± 0.25
MQ-FastGCN	Batch	2.41 ± 0.25	2.30 ± 0.30	2.05 ± 0.38	3.35 ± 0.42
	Training	21868.93 ± 165.25	16377.15 ± 150.35	31989.47 ± 195.50	34383.46 ± 215.40
	Metrics	51.69 ± 0.32	25.40 ± 0.38	42.97 ± 0.30	25.81 ± 0.48
FastGCN+f	Batch	7.57 ± 0.45	7.90 ± 0.55	8.61 ± 0.52	9.36 ± 0.65
	Training	65406.74 ± 495.30	53655.51 ± 425.10	129790.42 ± 875.30	92650.35 ± 640.45
	Metrics	62.10 ± 0.65	24.46 ± 0.45	53.12 ± 0.50	32.32 ± 0.40
MQ-FastGCN+f	Batch	2.30 ± 0.28	2.29 ± 0.32	1.87 ± 0.45	2.43 ± 0.55
	Training	20741.16 ± 170.20	16211.68 ± 155.50	29163.13 ± 205.75	25026.60 ± 185.40
	Metrics	60.40 ± 0.32	23.16 ± 0.35	52.52 ± 0.45	31.22 ± 0.50
FastGCN+d	Batch	7.45 ± 0.45	7.66 ± 0.40	8.49 ± 0.58	9.38 ± 0.55
	Training	64367.37 ± 470.50	52058.51 ± 400.50	128041.19 ± 860.25	92877.09 ± 645.35
	Metrics	61.78 ± 0.60	24.14 ± 0.55	54.74 ± 0.60	32.27 ± 0.45
MQ-FastGCN+d	Batch	2.33 ± 0.28	2.32 ± 0.32	1.93 ± 0.42	2.53 ± 0.50
	Training	20934.90 ± 180.20	16419.90 ± 165.50	30123.73 ± 215.00	26154.32 ± 150.00
	Metrics	59.98 ± 0.35	22.74 ± 0.40	54.04 ± 0.50	30.97 ± 0.55
FastGCN+f+d	Batch	7.45 ± 0.40	7.66 ± 0.35	8.49 ± 0.52	9.38 ± 0.50
	Training	64367.37 ± 455.35	52058.51 ± 385.25	**128041.19 ± 825.50**	92877.09 ± 620.15
	Metrics	61.78 ± 0.55	24.14 ± 0.50	**54.74 ± 0.50**	32.27 ± 0.40
MQ-FastGCN+f+d	Batch	2.31 ± 0.25	2.29 ± 0.28	1.88 ± 0.35	2.50 ± 0.42
	Training	20793.48 ± 165.25	16162.30 ± 150.35	**29399.34 ± 210.50**	25782.21 ± 185.25
	Metrics	60.18 ± 0.30	22.94 ± 0.35	**54.14 ± 0.40**	31.27 ± 0.50

where $P|E|$ represents the staleness cost, which increases linearly with P and is proportional to the number of edges $|E|$. The term $\frac{1}{P}\frac{|V|}{|\mathcal{G}|}$ represents the synchronization cost, which decreases inversely with P and is proportional to the number of nodes $|V|$, divided by the number of GPUs $|\mathcal{G}|$. The constants α and β are weighting coefficients that control the relative contribution of each term.

To minimize $C(P)$, we differentiate $C(P)$ with respect to P and set $\frac{dC(P)}{dP} = 0$:

$$\frac{dC(P)}{dP} = \alpha|E| - \beta\frac{1}{P^2}\frac{|V|}{|\mathcal{G}|}$$

Rearranging terms gives:

$$\alpha|E| = \beta\frac{1}{P^2}\frac{|V|}{|\mathcal{G}|}$$

Multiplying through by P^2 yields:

$$P^2 = \frac{\beta}{\alpha}\frac{|V|}{|\mathcal{G}|\,|E|}$$

Taking the square root of both sides gives:

$$P = \sqrt{\frac{\beta}{\alpha}}\sqrt{\frac{|V|}{|\mathcal{G}|\,|E|}}$$

Here, $\sqrt{\frac{\beta}{\alpha}}$ is treated as a constant scaling factor k. For simplicity, normalize $k = 1$, resulting in:

$$P = \frac{\sqrt{|V|}}{\sqrt{|\mathcal{G}|\,|E|}}$$

To ensure P is a discrete value suitable for implementation, apply the ceiling function:

$$P_{ms} = \left\lceil \frac{\sqrt{|V|}}{\sqrt{|\mathcal{G}|\,|E|}} \right\rceil$$

This derivation shows that P_{ms} dynamically adapts to the structure of the graph G ($|V|$ and $|E|$) and the number of GPUs ($|\mathcal{G}|$), ensuring a balance between staleness and synchronization costs.

■

TABLE 13. **Results for batch time (ms), training time (ms), and evaluation metrics (%) for LADIES, MQ-LADIES, and their enhanced variants on four-GPU configuration.**

	Benchmark	ogbn-proteins	ogbn-arxiv	Reddit	ogbn-products
LADIES	Batch	6.46 ± 0.28	10.18 ± 0.38	8.85 ± 0.45	8.37 ± 0.42
	Training	55798.51 ± 485.20	126832.24 ± 1025.75	139342.54 ± 905.80	123057.34 ± 925.35
	Metrics	69.30 ± 0.55	62.34 ± 0.65	75.70 ± 0.50	54.00 ± 0.60
MQ-LADIES	Batch	1.97 ± 1.45	2.91 ± 1.20	1.88 ± 0.50	2.11 ± 0.65
	Training	17756.09 ± 110.25	37821.10 ± 165.45	30600.62 ± 125.80	32376.30 ± 165.60
	Metrics	67.50 ± 0.25	61.04 ± 0.65	75.10 ± 0.37	53.00 ± 0.95
LADIES+f	Batch	6.81 ± 0.30	10.44 ± 0.40	11.00 ± 0.50	8.18 ± 0.35
	Training	58849.62 ± 520.35	102393.24 ± 850.40	173280.67 ± 1020.25	120272.43 ± 900.20
	Metrics	68.47 ± 0.45	64.10 ± 0.55	91.08 ± 0.50	62.31 ± 0.55
MQ-LADIES+f	Batch	2.05 ± 1.25	2.94 ± 1.00	2.31 ± 0.75	2.04 ± 0.50
	Training	18479.93 ± 125.72	30093.14 ± 160.65	37667.94 ± 145.50	31208.25 ± 180.40
	Metrics	66.97 ± 0.95	63.00 ± 0.45	90.58 ± 0.50	61.51 ± 0.35
LADIES+d	Batch	11.90 ± 0.50	12.11 ± 0.40	9.47 ± 0.60	8.31 ± 0.45
	Training	102820.34 ± 810.35	163524.31 ± 1325.40	213050.34 ± 1650.50	122178.21 ± 1050.35
	Metrics	69.29 ± 0.55	62.30 ± 0.45	88.44 ± 0.50	54.89 ± 0.60
MQ-LADIES+d	Batch	3.66 ± 1.00	3.56 ± 0.90	2.06 ± 0.50	2.16 ± 0.30
	Training	33037.28 ± 150.50	50276.67 ± 190.85	47833.83 ± 175.65	32997.83 ± 180.90
	Metrics	67.59 ± 1.00	60.90 ± 0.85	87.74 ± 0.50	53.79 ± 0.45
LADIES+f+d	Batch	11.70 ± 0.50	13.03 ± 0.65	11.00 ± 0.55	**8.36 ± 0.45**
	Training	101075.36 ± 810.45	129564.43 ± 1250.35	**247618.18 ± 1620.65**	**122857.42 ± 1035.25**
	Metrics	68.07 ± 0.45	60.00 ± 0.55	**89.43 ± 0.55**	62.60 ± 0.45
MQ-LADIES+f+d	Batch	3.63 ± 1.20	3.88 ± 1.35	2.44 ± 0.40	**2.23 ± 0.85**
	Training	32675.67 ± 145.85	40188.03 ± 175.90	**56834.54 ± 150.55**	**34087.17 ± 125.50**
	Metrics	66.47 ± 0.25	58.80 ± 0.85	**88.83 ± 0.60**	61.60 ± 0.20

APPENDIX B
RESULTS COMPARISON WITH SoTA FOR THREE, AND FOUR GPUs

Tables 8 to 13 present the comparison results for three and four GPU configurations, demonstrating MQ-GNN's scalability and efficiency. The results highlight training speedup, batch time improvements, and the impact of staleness on evaluation metrics, showing how MQ-GNN balances performance and accuracy through periodic synchronization.

REFERENCES

[1] I. Chami, S. Abu-El-Haija, B. Perozzi, C. Ré, and K. Murphy, "Machine learning on graphs: A model and comprehensive taxonomy," *J. Mach. Learn. Res.*, vol. 23, no. 1, pp. 3840–3903, 2022.

[2] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, pp. 1025–1035.

[3] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lió, and Y. Bengio, "Graph attention networks," *Stat*, vol. 1050, p. 48550, Jan. 2017.

[4] D. Zou, Z. Hu, Y. Wang, S. Jiang, Y. Sun, and Q. Gu, "Layer-dependent importance sampling for training deep and large graph convolutional networks," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 1–11.

[5] J. Chen, T. Ma, and C. Xiao, "FastGCN: Fast learning with graph convolutional networks via importance sampling," in *Proc. Int. Conf. Learn. Represent.*, 2018, pp. 1–15.

[6] A. Strokach, D. Becerra, C. Corbi-Verge, A. Perez-Riba, and P. M. Kim, "Fast and flexible protein design using deep graph neural networks," *Cell Syst.*, vol. 11, no. 4, pp. 402–411, Oct. 2020.

[7] W. Fan, Y. Ma, Q. Li, Y. He, E. Zhao, J. Tang, and D. Yin, "Graph neural networks for social recommendation," in *Proc. World Wide Web Conf.*, May 2019, pp. 417–426.

[8] N. Park, A. Kan, X. L. Dong, T. Zhao, and C. Faloutsos, "Estimating node importance in knowledge graphs using graph neural networks," in *Proc. 25th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Jul. 2019, pp. 596–606.

[9] S. Gandhi and A. P. Iyer, "P3: Distributed deep graph learning at scale," in *Proc. 15th Symp. Operating Syst. Design Implement.*, 2021, pp. 551–568.

[10] J. Thorpe, Y. Qiao, J. Eyolfson, S. Teng, G. Hu, Z. Jia, J. Wei, K. Vora, R. Netravali, M. Kim, and G. Xu, "Dorylus: Affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads," in *Proc. 15th USENIX Symp. Operating Syst. Design Implement.*, Jan. 2021, pp. 495–514.

[11] I. F. Ilyas, T. Rekatsinas, V. Konda, J. Pound, X. Qi, and M. Soliman, "Saga: A platform for continuous construction and serving of knowledge at scale," in *Proc. Int. Conf. Manage. Data*, Jun. 2022, pp. 2259–2272.

[12] Z. Lin, C. Li, Y. Miao, Y. Liu, and Y. Xu, "PaGraph: Scaling GNN training on large graphs via computation-aware caching," in *Proc. 11th ACM Symp. Cloud Comput.*, Oct. 2020, pp. 401–415.

[13] 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. (TOPC)*, vol. 5, no. 3, pp. 1–39, 2019.

[14] Z. Zhang, P. Cui, and W. Zhu, "Deep learning on graphs: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 1, pp. 249–270, Mar. 2020.

[15] J. Chen, J. Zhu, and L. Song, "Stochastic training of graph convolutional networks with variance reduction," in *Proc. Int. Conf. Mach. Learn.*, Jan. 2017, pp. 942–950.

[16] W. Huang, T. Zhang, Y. Rong, and J. Huang, "Adaptive sampling towards fast graph representation learning," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 31, 2018, pp. 4563–4572.

[17] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "GraphSAINT: Graph sampling based inductive learning method," in *Proc. Int. Conf. Learn. Represent.*, 2020, pp. 1–19. [Online]. Available: https://openreview.net/forum?id=BJe8pkHFwS

[18] J. Dong, D. Zheng, L. F. Yang, and G. Karypis, "Global neighbor sampling for mixed CPU-GPU training on giant graphs," in *Proc. 27th ACM SIGKDD Conf. Knowl. Discovery Data Mining*. New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 289–299, doi: 10.1145/3447548.3467437.

[19] K. Huang, J. Zhai, Z. Zheng, Y. Yi, and X. Shen, "Understanding and bridging the gaps in current GNN performance optimizations," in *Proc. 26th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, Feb. 2021, pp. 119–132.

[20] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken, "Improving the accuracy, scalability, and performance of graph neural networks with roc," *Proc. Mach. Learn. Syst.*, vol. 2, pp. 187–198, Mar. 2020.

[21] D. Zheng, M. Wang, Q. Gan, X. Song, Z. Zhang, and G. Karypis, "Scalable graph neural networks with deep graph library," in *Proc. 14th ACM Int. Conf. Web Search Data Mining*, Mar. 2021, pp. 1141–1142, doi: 10.1145/3437963.3441663.

[22] S. W. Min, K. Wu, S. Huang, M. Hidayetoğlu, J. Xiong, E. Ebrahimi, D. Chen, and W.-M. Hwu, "PyTorch-direct: Enabling GPU centric data access for very large graph neural network training with irregular accesses," 2021, *arXiv:2101.07956*.

[23] T. Kaler, N. Stathas, A. Ouyang, A.-S. Iliopoulos, T. Schardl, C. E. Leiserson, and J. Chen, "Accelerating training and inference of graph neural networks with fast sampling and pipelining," *Proc. Mach. Learn. Syst.*, vol. 4, pp. 172–189, Apr. 2022.

[24] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 32, no. 1, pp. 4–24, Jan. 2021.

[25] D. Zheng, X. Song, C. Yang, D. LaSalle, and G. Karypis, "Distributed hybrid CPU and GPU training for graph neural networks on billion-scale heterogeneous graphs," in *Proc. 28th ACM SIGKDD Conf. Knowl. Discovery Data Mining*, Aug. 2022, pp. 4582–4591.

[26] L. Van der Perre, L. Liu, and E. G. Larsson, "Efficient DSP and circuit architectures for massive MIMO: State of the art and future directions," *IEEE Trans. Signal Process.*, vol. 66, no. 18, pp. 4717–4736, Sep. 2018.

[27] G. Sheng, J. Su, C. Huang, and C. Wu, "MSPipe: Efficient temporal GNN training via staleness-aware pipeline," in *Proc. 30th ACM SIGKDD Conf. Knowl. Discovery Data Mining*, Aug. 2024, pp. 2651–2662.

[28] Y. Chen, T. Xu, D. Hakkani-Tür, D. Jin, Y. Yang, and R. Zhu, "Calibrate and debias layer-wise sampling for graph convolutional networks," *Trans. Mach. Learn. Res.*, vol. 1, pp. 1–32, Jan. 2022. [Online]. Available: https://openreview.net/forum?id=JyKNuoZGux

[29] J. Mohoney, R. Waleffe, Y. Xu, T. Rekatsinas, and S. Venkataraman, "Marius: Learning massive graph embeddings on a single machine," in *Proc. 15th USENIX Symp. Operating Syst. Design Implement.*, 2021, pp. 533–549.

[30] J. Chen, Z. Chen, and X. Qian, "GNNPipe: Scaling deep GNN training with pipelined model parallelism," 2023, *arXiv:2308.10087*.

[31] C. Wan, Y. Li, C. Wolfe, A. Kyrillidis, N. S. Kim, and Y. Lin, "PipeGCN: Efficient full-graph training of graph convolutional networks with pipelined feature communication," in *Proc. 10th Int. Conf. Learn. Represent.*, 2022, pp. 1–24.

[32] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch geometric," 2019, *arXiv:1903.02428*.

[33] P. S. Efraimidis and P. G. Spirakis, "Weighted random sampling with a reservoir," *Inf. Process. Lett.*, vol. 97, no. 5, pp. 181–185, Mar. 2006.

[34] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *Proc. Int. Conf. Learn. Represent.*, 2016, pp. 1–14.

[35] 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 *Proc. USENIX Annu. Tech. Conf.*, Jul. 2019, pp. 443–458.

[36] 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, no. 8, pp. 1–39, Aug. 2024.

[37] 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," in *Proc. 34th Int. Conf. Neural Inf. Process. Syst.* Red Hook, NY, USA: Curran Associates, 2020, pp. 22118–22133.

IRFAN ULLAH received the master's degree in computer science from the National University of Sciences and Technology (NUST), Pakistan. He is currently pursuing the Ph.D. degree in computer science with Kyung Hee University, South Korea.

Previously, he held research and teaching positions with NUST, Federal Urdu University of Arts, Science, and Technology (FUUAST), and the multi-national company Knowledge Platform, Pakistan. He is also a Research Assistant with the Department of Computer Science and Engineering, Kyung Hee University. His work has been published in high-impact journals and international conferences and he has authored several patents. He has contributed to multiple research projects on scalable GNN training, memory-aware computing, distributed graph learning, and social media data analysis. He has also been actively involved in open-source projects. He has delivered talks and tutorials on programming and development, natural language processing, distributed computing, machine learning, and machine learning with graphs. His research interests include graph neural networks, machine learning, deep learning, big data analytics, distributed computing, social computing, natural language processing, operating system design, and memory system optimization.

YOUNG-KOO LEE (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer science from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, South Korea, in 1992, 1994, and 2002, respectively.

From 2002 to 2004, he was a Postdoctoral Researcher with the University of Illinois at Urbana–Champaign, and a Postdoctoral Fellow with the Advanced Information Technology Research Center (AITrc), KAIST. In 2004, he joined Kyung Hee University, Global Campus, South Korea, where he has held various academic positions. He was an Assistant Professor from 2004 to 2010, an Associate Professor from 2010 to 2015, and has been a Professor since 2015 with the College of Software. He has published numerous research papers in top-tier journals and conferences. He has actively contributed to advancing data science and graph neural network (GNN)-based learning methodologies. His research interests include GNNs, data mining, online analytical processing, query optimization, and big data processing.

Prof. Lee received recognition for his contributions to large-scale graph processing and deep-learning applications.