Efficient Distributed Graph Neural Network Training With Source Chunking and Moving Aggregation

Wenjie Huang, Tongya Zheng, Rui Wang, Tongtian Zhu, Bingde Hu, Shuibing He, *Member, IEEE*, Mingli Song, *Senior Member, IEEE*, Xinyu Wang, Sai Wu, and Chun Chen

***Abstract*—Graph neural networks (GNNs) are effective models for analyzing graph-structured data, but encounter challenges when training on large distributed graphs. Existing GNN training frameworks use sampling parallelism and historical embedding methods to support distributed training and enhance efficiency. However, these methods suffer from issues like stale historical embeddings, imbalanced communication messages, and redundant storage and computation costs. In this paper, we present Emma, a distributed GNN training framework that incorporates source node centric chunking for frequent updates of embeddings and balanced communication, as well as a moving message aggregation technique to boost training efficiency and reduce storage costs. Experimental results show that Emma significantly enhances training efficiency by reducing computation and communication overhead, leading to a notable speedup while maintaining convergence accuracy compared to state-of-the-art distributed GNN training methods.**

***Index Terms*—Graph neural networks (GNNs), distributed GNN training, graph message aggregation, historical embedding.**

I. Introduction

GRAPH Neural Networks (GNNs) are popular for analyzing graph data by capturing node relationships. Techniques like message passing and neighborhood aggregation allow GNNs to learn node representations effectively by combining from neighboring information [1]. They have been used successfully in tasks such as node classification [2], [3], [4], graph classification [5], [6], [7], and edge prediction [8], [9], [10], [11].

Various *distributed partition parallel training* methods have been created to enhance GNN training efficiency on large graphs. These methods partition the graph data into subgraphs and distribute them among multiple GPUs or machines for parallel training. Effective communication between these devices is crucial for coordinating the training process efficiently. The vanilla full-batch GNN training methods, where all nodes in the graph are trained in each iteration, can be computationally and communicationally expensive, particularly in situations with numerous edge connections between subgraphs.

To tackle these issues, various *sampling parallel training* methods have been developed to enhance GNN training efficiency [3], [12], [13], which sample mini-batches from the entire graph and train each mini-batch on individual GPUs, thus improving training efficiency. However, sampling methods may introduce variance that can impact the model's accuracy. Recent research has leveraged *historical embeddings* to address concerns related to distributed communication and sampling variance, such as using local historical embeddings to reduce communication [14], [15], incorporating historical embeddings for unsampled nodes to mitigate variance [16], [17], and training exclusively with historical embeddings [18], [19], [20].

Sampling parallelism and historical embedding-based methods have enhanced GNN training but also introduced new challenges. One such challenge stems from their used *target-node centric sampling* method, which involves sampling target nodes, updating them by aggregating messages from their in-neighbors across all partitions in each GNN layer recursively. Unfortunately, this approach has led to issues concerning stale historical embeddings [15], [18] and imbalanced communication costs [14], [21], due to low sample utilization and unbounded pulling messages, ultimately impact the overall effectiveness and efficiency of GNN training.

Furthermore, current historical embedding-based methods use historical embeddings as the most recent embeddings for remote aggregation, resulting in costly storage overhead, as numerous historical embeddings of all nodes in all layers need to be stored [22]. Additionally, the repetitive aggregation of these historical embeddings in each training epoch consumes computational resources and impedes training efficiency [16].

To tackle these issues and delve deeper into the usage of historical embeddings in distributed GNN training, we propose a novel distributed GNN training framework *Emma*. It integrates

Received 8 October 2024; revised 11 June 2025; accepted 5 September 2025. Date of publication 24 September 2025; date of current version 10 November 2025. This work was supported in part by the Starry Night Science Fund of Zhejiang University Shanghai Institute for Advanced Study under Grant SN-ZJU-SIAS-001, in part by the Hangzhou Joint Fund of the Zhejiang Provincial Natural Science Foundation of China under Grant LHZSD24F020001, and in part by the Zhejiang Province High-Level Talents Special Support Program "Leading Talent of Technological Innovation of Ten-Thousands Talents Program" under Grant 2022R52046. Recommended for acceptance by R. Akbarinia. *(Corresponding author: Rui Wang.)*

Wenjie Huang, Tongtian Zhu, Bingde Hu, Shuibing He, Xinyu Wang, and Chun Chen are with Zhejiang University, Hangzhou 310027, China (e-mail: wjie@zju.edu.cn; raiden@zju.edu.cn; tonyhu@zju.edu.cn; heshuibing@zju.edu.cn; wangxinyu@zju.edu.cn).

Tongya Zheng is with the Big Graph Center, School of Computer and Computing Science, Hangzhou City University, Hangzhou 310015, China, and also with the Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, Hangzhou 310053, China (e-mail: tyzheng@zju.edu.cn).

Rui Wang, Mingli Song, and Sai Wu are with Zhejiang University, Hangzhou 310027, China, and also with the Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, Hangzhou 310053, China (e-mail: rwang21@zju.edu.cn; brooksong@zju.edu.cn; wusai@zju.edu.cn).

Digital Object Identifier 10.1109/TKDE.2025.3613787

source node-centric chunking and *moving message aggregation* techniques to reduce computational and communication costs, while ensuring convergence accuracy similar to full-batch GNN training. Our key contributions can be outlined as follows:

- We comprehensively analyze the limitations in existing sampling parallelism and historical embedding-based distributed GNN training methods, which includes issues of embedding staleness, imbalanced communication, and excessive storage and computation costs. We delve into the specific causes behind these issues and assess their significance and impact.
- We introduce a *source node-centric chunking* approach that incorporates historical embeddings in GNN training, which divides the full-batch source nodes into multiple chunks, sequentially selecting each chunk, and pushing their latest embeddings to their out-neighboring nodes. This approach enables more frequent updates of historical embeddings, reducing staleness. Furthermore, it supports bounded pushing messages, enhancing communication balance and efficiency.
- We introduce a *moving message aggregation technique* to update target node embeddings incrementally. Using a moving average-based approach, we approximate incremental aggregation by combining historical aggregation values with newly received messages from source nodes, reducing computational and storage costs for message aggregation.
- Elaborate experiments are conducted on common GNN models to demonstrate the effectiveness and efficiency of *Emma*. The results indicate that *Emma* surpasses existing distributed GNN training methods, resulting in a notable improvement in training efficiency and impressive training accuracy. Our code is made publicly available at https://github.com/wjie98/EmmaGNN.

II. Background and Motivation

A. Graph Neural Networks

Message passing in GNNs: Graph neural networks (GNNs) perform non-linear message-passing operations on graph structures, drawing inspiration from message-passing neural networks (MPNN) [23]. MPNN breaks down GNN operations into three main stages: message generation, message aggregation, and node update, with their associated functions given below:

$$\begin{aligned}\mathbf{m}_{ij}^{(l)} &= \text{Message}_{\theta}^{(l)}\left(\mathbf{h}_i^{(l-1)},\ \mathbf{e}_{ij},\ \mathbf{h}_j^{(l-1)}\right),\\ \mathbf{z}_i^{(l)} &= \text{Aggregate}_{\theta}^{(l)}\left(\{\mathbf{m}_{ij}^{(l)},\ \forall j \in \mathcal{N}(i)\}\right),\\ \mathbf{h}_i^{(l)} &= \text{Update}_{\theta}^{(l)}\left(\mathbf{z}_i^{(l)},\ \mathbf{h}_i^{(l-1)}\right),\end{aligned} \tag{1}$$

where $\mathcal{N}(i)$ is the neighbors of node i, $\mathbf{h}_i^{(l)}$ is hidden states of node i at layer l, $\mathbf{e}_{ij}$ is edge features between nodes i and j, and $\mathbf{m}_{ij}^{(l)}$ is the message generated at layer l for edge $e(i,j)$.

Graph convolutional networks (GCNs) [2] and GraphSAGE [3] are common GNNs. GCNs use convolutional networks for message aggregation, while GraphSAGE employs neighbor sampling for mini-batch training. Graph attention network (GAT) [4] enhances GCNs by incorporating a self-attention mechanism for message aggregation, allowing nodes to assign weights to messages based on importance. These GNN models follow the MPNN pattern for message passing operations.

Distributed partition parallel training: As graph data and GNN model complexity grow, with real-world graphs potentially containing millions of nodes and billions of edges, a single GPU may struggle to handle them efficiently. Distributed computing is a viable solution to improve the training ability and efficiency of GNN models. One common approach for distributed GNN training is partition parallel training, where the graph is divided into multiple subgraphs by graph partition algorithms [24], [25], and each GPU is assigned to perform local message aggregation on a subgraph using the GNN replica. The model weights are synchronously updated through remote and local access, as depicted in Fig. 1(a). Various works have explored this method [14], [15], [22], [26], [27], [28], [29].

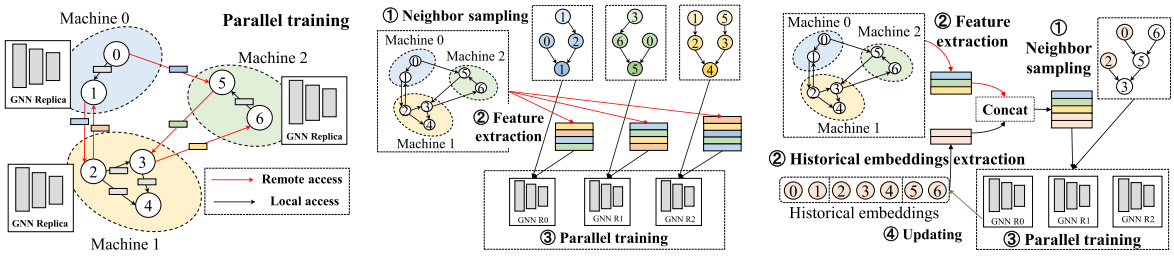
Distributed sampling parallel training: Distributed sampling parallel training integrates graph sampling techniques [3], [12], [13], [16], [30] with existing partition parallel methods [31], [32], [33], where each GPU independently samples a mini-batch of nodes from the graph data, extracts feature information related to the sampled nodes remotely, and conducts parallel GNN training across multiple GPUs, as depicted in Fig. 1(b). Using sampled mini-batches allows for more frequent updates to the model weights, improving training efficiency. However, challenges such as sampling variance and communication overhead for fetching raw features across machines need to be addressed. Additionally, concerns about the utilization of sampling nodes may impact convergence accuracy or require a larger number of samplers for each training epoch.

Historical embedding enhanced GNN training: Historical embeddings have significantly enhanced GNN training in recent years. In distributed partition parallel training, locally preserved historical embeddings can eliminate the need for frequent distributed data transfers [15], [28] or the concurrent execution of computation and communication tasks [14]. For neighbor sampling methods, historical embeddings can supplement unsampled nodes, reduce sampling variance, and improve training efficiency and accuracy [16], [20], [34], as shown in Fig. 1(c). Despite these benefits, using historical embeddings may introduce staleness issues, especially in sampling methods where only the historical embeddings of nodes within the mini-batch can be updated in each iteration.

B. Limitations in Distributed Training on Large Graphs

Existing sampling parallelism models typically utilize the *target node centric sampling* method, which involves sampling a batch of target nodes, pulling the embeddings of their in-neighbors from all partitions, and updating the embeddings of sampled target nodes by aggregating messages from their in-neighbors. This iterative process continues until the GNN models converge. While these methods have enhanced GNN training, they encounter difficulties when handling large graphs.

Limitation #1: Stale historical embeddings: Fresh historical embeddings are crucial for historical embedding-based GNN



(a) Partition parallel training (b) Sampling parallel training (c) Historical embedding based training

Fig. 1. Comparison of different distributed GNN training frameworks: (a) Partition parallel training involves distributing graph data across multiple GPUs and requires frequent inter-partition communication to access neighboring node embeddings. (b) Sampling parallel training uses subgraph sampling to create mini-batches for data-parallel training across GPUs, reducing the need for inter-partition communication. However, it may introduce sampling variance issues by dropping embeddings of unsampled neighbors. (c) Historical embedding-based training employs smaller sampling subgraphs with cached historical embeddings to supplement unsampled neighbors and reduce sampling variance. While effective in mitigating sampling variance, this method may encounter challenges related to historical embedding staleness.

training. However, the target node centric sampling method only updates historical embeddings of sampled target nodes within the mini-batch once per training iteration. This leads to low sample utilization and infrequent updates to historical embeddings, resulting in issue of stale historical embeddings. As a consequence, outdated node embeddings cause the model to learn from obsolete information, perpetuating a feedback loop of suboptimal embeddings propagating through the computation graph, causing potential convergence issues, such as overfitting and trapping in local optima [18].

To investigate the severity of stale historical embeddings issue and its impact on model weight updates, we performed experiments to evaluate the effects of node-wise sampling, layer-wise sampling, and our suggested layer-wise source node centric chunking method (see Section III-A) on the staleness of historical embeddings using the Reddit dataset. To quantify staleness, we compared the model weight version used for calculating historical embeddings with the most recent weight version, with smaller differences indicating better performance. Note that we used model weight comparison to assess staleness instead of evaluating the similarity of embeddings. This is because both model weight and historical embeddings are iteratively and alternately updated using each other, and embeddings computed with up-to-date model weights provide more accurate gradient estimation. While similarity metrics may suggest the closeness of embeddings, they could potentially mask suboptimal local minima that impede global optimization [15]. We set the batch size as 8192 and run a 3-layer GNN model. In node-wise sampling, each node sampled 8 neighbors, while in layer-wise sampling, each GNN layer sampled nodes equal to the batch size. The results, depicted in Fig. 2, reveal that after three epochs of training using the target node centric sampling method, certain historical embeddings were notably outdated, with variances exceeding 30 steps for node-wise sampling and over 70 steps for layer-wise sampling. In contrast, our node centric chunking approach exhibited minimal staleness, with only a handful of nodes having historical embeddings outdated by more than 3 steps, attributed to its more frequent node update strategy.

Limitation #2: Imbalanced communication messages: In the target node centric sampling method, each sampled target node needs to recursively fetch messages from its in-neighbors in each

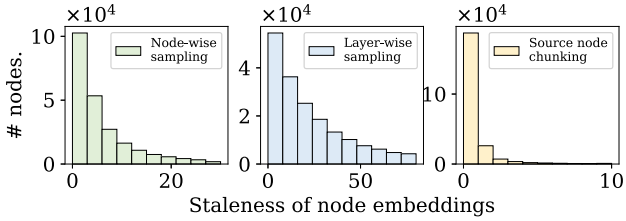


Fig. 2. The staleness histogram of the node embeddings at the last GNN layer after training for 3 epochs on Reddit dataset.

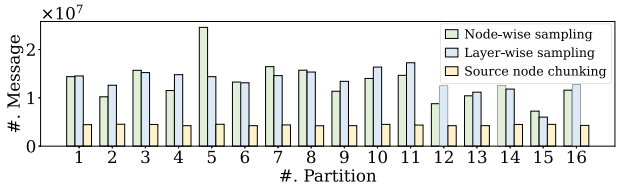


Fig. 3. The number of communicated messages sent by different partitions during a single communication cycle.

GNN layer, which includes features and historical embeddings of neighboring nodes from other partitions (refer to Fig. 1). This can lead to an unbounded retrieval of messages due to the neighbor explosion problem. Real-world graphs exhibit locality characteristics and a power law distribution of node degrees, resulting in uneven distribution of neighboring nodes of sampled target nodes across different partitions. This exacerbates the issue of imbalanced cross-partition communication messages, thereby affecting training efficiency.

To assess the severity of imbalanced communication messages issue, we conducted tests on communication load balancing with various sampling methods in distributed clusters, using the same experimental setup above. We partition the Reddit graph into 16 partitions, and use 16 GPUs for parallel GNN training. We tracked the number of communicated messages (each message representing a node embedding) sent by different partitions during a single communication cycle, and the results are presented in Fig. 3. The analysis revealed highly imbalanced communicated message distributions for both node-wise sampling and layer-wise sampling, with the maximum value

being 3.39× and 3.04× greater than the minimum value, respectively. In contrast, our source node centric chunking method demonstrated a balanced distribution of communicated messages among the 16 partitions, with the difference between the maximum and minimum values being less than 38%.

Limitation #3: Redundant storage and computation costs: While using local cached historical embeddings can help reduce communication overhead by substituting remote neighbor messages from other partitions, it necessitates storing historical embeddings of all in-neighbors for all GNN layers within the target node centric sampling methods. This results in significant storage costs due to the large number of historical embeddings that need to be stored [22]. For example, when partitioning the Reddit graph into 16 partitions, each target node has an average of 6.33 intra-partition in-neighboring nodes, resulting in the need to store 6.33 historical embeddings for each target node. This leads to much higher storage expenses compared to storing embeddings of target nodes alone. Additionally, relying on cached historical embeddings for message aggregation calculations in each training epoch results in redundant aggregation of stale neighbor embeddings, consuming more computational resources and slowing down the model training process [16], [22].

III. Design of *Emma*

We aim to overcome the limitations mentioned above and enhance the computation and communication efficiency of distributed GNN training while maintaining the model convergence accuracy as full-batch training. To achieve this goal, we present *Emma*, a distributed GNN training framework that incorporates two innovative techniques: *source node centric chunking* and *moving message aggregation*. Instead of sampling target nodes and pulling embedding messages from their in-neighbors, the source node centric chunking approach involves chunking the entire graph nodes into multiple chunks and each time sequentially selecting one chunk, and then pushing and transmitting their fresh embeddings to their out-neighbors for subsequent message aggregation. Subsequently, the framework utilizes the moving message aggregation technique to update the embeddings of target nodes, leveraging both the historical embeddings of target nodes and newly received messages from the source nodes. Consequently, *Emma* improves embedding staleness by enabling more frequent updates of historical embeddings, as one source node can contribute to multiple target node updates. Moreover, it also reduces the computation, communication, and storage costs for computing node representations. The overview of *Emma* is shown in Fig. 4. Next, we introduce the proposed source node centric chunking and moving message aggregation techniques in detail.

A. Source Node Centric Chunking

Skewed number of source and target nodes: In distributed GNN training graphs, there is often a higher number of source nodes compared to target nodes in each partition, especially for hub nodes with many neighbors. This imbalance becomes more pronounced as the number of partitions increases. To verify this observation, we utilized the widely used METIS graph partitioning algorithm [24] to divide the Reddit dataset into varying numbers of partitions. We then calculated the average number of target nodes and their corresponding source neighbors in each partition, as shown in Table I. The results reveal a linear decrease in the number of target nodes with an increasing number of partitions. However, the number of source nodes does not decrease proportionally due to the presence of common neighbors among target nodes across partitions, leading to the duplication of source nodes on multiple partitions. Current target node-centric sampling-based GNN training methods store historical embeddings on source nodes for pulling to sampled target nodes. Target nodes sharing common neighbors in different batches result in redundant message fetching and less frequent updates, resulting in stale embeddings. The uneven and unrestricted fetching of messages from numerous source nodes also contributes to imbalanced communication costs, as discussed in Section II-B.

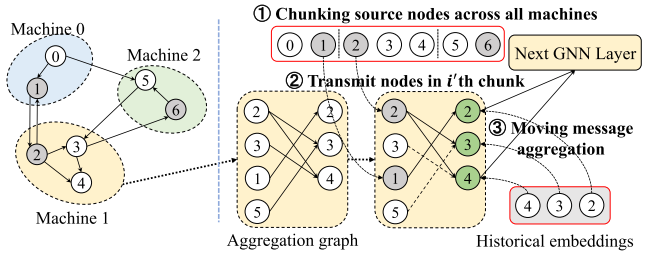


Fig. 4. Overview of the training framework Emma. The graph is divided into three partitions on the left side, with each machine (GPU) loading a complete partition and communicating with others via collective operations. The right side illustrates the execution process on Machine 1 for a single layer of the GNN. Nodes in these partitions are divided into B chunks, such as chunk 0 ($\{1, 2, 6\}$) and chunk 1 ($\{0, 3, 4, 5\}$). The training process involves three main steps: (1) In the current iteration, chunk 0 ($\{1, 2, 6\}$) are selected as *source nodes*, while the other chunk is deferred for subsequent iterations. (2) The selected source nodes send their updated embeddings to their out-neighbors, known as *target nodes*, such as $\{2, 3, 4\}$ on Machine 1. (3) The embeddings of the target nodes are updated using the latest embeddings received from the source nodes and the locally stored historical embeddings through moving message aggregation. This process is repeated for each layer of the GNN, completing the entire training process.

TABLE I
The Average Number of Target Nodes and Their Source Neighbors of Each Partition on Reddit

# Partitions.	2	4	6	8	10	12	14	16
# target.($\times 10^3$)	116.48	58.24	38.83	29.12	23.30	19.41	16.64	14.56
# source.($\times 10^3$)	190.08	148.19	122.58	111.97	105.45	99.84	91.88	92.21
# source./# target.	1.63	2.54	3.16	3.85	4.53	5.14	5.52	6.33

Source node centric chunking based GNN training: Acknowledging the discrepancy in source and target nodes per partition and aiming to overcome challenges in existing target node-centric sampling-based methods, we propose a novel *source node-centric chunking strategy*. In this approach, the full-batch graph source nodes are partitioned into multiple chunks. During each iteration, a chunk of source nodes $\mathcal{S}_b$ is selected from the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, and their updated embeddings are pushed to their out-neighbors, necessitating cross-partition communication for transmitting these embedding messages. Subsequently, target nodes $\mathcal{V}_b = \{i | j \in \mathcal{S}_b, (j, i) \in \mathcal{E}\}$ are updated using the

embeddings of their neighboring source nodes in the current iteration. The specific chunking and updating process is as follows. First, the nodes of the entire graph are evenly divided into B chunks, denoted by source nodes,

$$s = \left[s_1, s_2, \ldots, s_{|\mathcal{V}|}\right], \text{ where } s_j \sim \mathcal{U}(1, B),$$
$$\mathcal{S}_b = \{j \,|s_j = b, \forall j \in \mathcal{V}\}, \tag{2}$$

where B is the number of chunks and $\mathcal{U}(1, B)$ is the uniform distribution ranging from 1 to B. In a distributed environment, we consider a partition $\mathcal{G}_k = \{\mathcal{V}_k, \mathcal{E}_k\}$ for the k-th partition, where $\mathcal{E}_k$ is the induced edge set of the disjoint node set $\mathcal{V}_k$. We broadcast the source nodes $\mathcal{S}_b$ to all partitions and obtain the target nodes $\mathcal{V}_{bk} \subseteq \mathcal{V}_k$ at each partition as

$$\mathcal{V}_{bk} = \{i \,|j \in \mathcal{S}_b, (j, i) \in \mathcal{E}_k\}. \tag{3}$$

The generated B chunks are used for iterative training, which updates the model weights B times within B iterations. The communication volume required for each iteration is reduced to $\frac{1}{B}$ of the original amount. Moreover, since the data transmitted by each source node to neighboring partitions remains constant, a simple random chunking strategy can ensure that communication volume is approximately uniform across the chunks without incurring additional overhead. The amount of data sent by the source nodes is positively correlated with the node degree. Consequently, the probability that the top k source nodes with the highest communication volume are assigned to the same chunk is less than $(\frac{1}{B})^k$. This can be approximated as different source nodes with varying communication loads being uniformly distributed across multiple chunks, resulting in balanced communication across chunks.

Benefits of source node-centric chunking: Source node-centric chunking optimizes historical embedding-based distributed GNN training by balancing communication efficiency and embedding freshness. To enhance communication efficiency, a global chunking strategy randomly divides all source nodes into B chunks across the graph, ensuring uniform subsets and reducing cross-device communication costs through synchronized seeds and localized computation. Synchronized random seeds enable independent chunk membership computation, while localized computation allows each partition to determine target nodes without cross-partition coordination. For embedding freshness, source node-centric chunking reduces staleness by enabling more frequent updates of target node embeddings within each training epoch. Every source node is assigned to one chunk, processed sequentially within B iterations to guarantee updates for all out-neighbors. Self-loop edges and messages ensure frequent updates for nodes with low in-degrees. This strategy effectively manages staleness for all nodes, with low in-degree nodes benefiting from frequent self-updates and high in-degree nodes utilizing adaptive decay to maintain embedding freshness.

B. Moving Message Aggregation

To address the issue of redundant computation and high storage costs associated with neighbor aggregation by reusing historical embeddings, we further propose a novel *moving message aggregation method*. This method aims to minimize these costs by leveraging the historical embeddings and approximating an incremental neighbor aggregation process:

$$\mathbf{z}_i = \bar{\mathbf{z}}_i \ominus \{\bar{\mathbf{m}}_{ij}\} \oplus \{\mathbf{m}_{ij}\}, \text{ for } j \in \hat{\mathcal{N}}(i), \tag{4}$$

where $\mathbf{m}_{ij}$ denotes the message generated from source nodes, $\bar{\mathbf{m}}_{ij}$ represents the message received from previous chunks, $\bar{\mathbf{z}}_i$ is the historical aggregated result for node i, and $\hat{\mathcal{N}}(i)$ indicates the set of neighbors of node i involved in cross-partition communication in the current chunk. This equation updates the historical aggregated result of the target node by replacing old messages with new messages from source nodes. The $\oplus$ operator removes outdated messages, while the $\ominus$ operator incorporates new ones. This optimization leverages historical aggregated result to reduce redundant computation and storage costs in the neighbor aggregation process.

While the incremental aggregation technique offers a direct and efficient way to update aggregated result without redundant computations or excessive storage, it does have limitations. It can result in unstable computation outcomes due to accumulated floating-point errors during incremental updates. Furthermore, it requires storing historical versions of neighbor messages, adding complexity and challenges to the neighbor aggregation process. Therefore, we introduce a decay coefficient β for the historical results of each node:

$$\mathbf{z}_i = \beta_i \bar{\mathbf{z}}_i + \text{Aggr}\{\mathbf{m}_{ij}, \ \forall j \in \hat{\mathcal{N}}(i)\}, \tag{5}$$

where $\deg_{\text{in}}(i)$ represents the in-degree of node i. The message aggregation operation $\text{Aggr}\{\mathbf{m}_{ij}, \ \forall j \in \hat{\mathcal{N}}(i)\}$ combines the new messages $\mathbf{m}_{ij}$ from the neighbors of node i that are involved in cross-partition communication.

Traditional moving average methods typically use a fixed decay coefficient β, which can lead to issues. First, the source node partitioning strategy could render certain target nodes as isolated (i.e., $|\hat{\mathcal{N}}(i)| = 0$), where neighbor messages carry no informative content. A fixed β would continuously integrate these non-informative embeddings into node representations, potentially degrading the model. Second, the source node partitioning causes uneven message counts across aggregation steps for each node, making fixed β suboptimal for handling such dynamic variations. To address these challenges, we derive an adaptive decay coefficient through variance stability analysis. Assuming that both new messages and historical embeddings follow zero-mean normal distributions with unit variance, we design the decay coefficient β to maintain variance consistency and prevent model divergence:

$$\beta_i = 1 - \frac{1}{\deg_{\text{in}}(i)} \cdot |\hat{\mathcal{N}}(i)| \tag{6}$$

This formulation ensures that the combination of historical embeddings ($\beta_i \bar{\mathbf{z}}_i$) and new messages ($\text{Aggr}\{\mathbf{m}_{ij}\}$) maintains stable variance characteristics. The term $|\hat{\mathcal{N}}(i)|/\deg_{\text{in}}(i)$ automatically adjusts the decay strength based on the proportion of active neighbors in each chunk, achieving a balanced incorporation of historical information and new messages.

Integration with generic GNN models: The moving message aggregation method can be seamlessly integrated with various

message aggregation functions, including $\mathrm{sum}(\cdot)$, $\mathrm{mean}(\cdot)$, and $\mathrm{attention}(\cdot)$, which are commonly used in GNN models. This flexibility enables the method to be easily incorporated into a wide range of GNN architectures. To showcase its versatility, we will illustrate how the moving message aggregation technique can be implemented in two popular GNN models: GCN and GAT. Utilizing the moving message aggregation paradigm, we can adapt the $\mathrm{sum}(\cdot)$ function of GCN [2] below:

$$\mathbf{z}_i = \beta_i \bar{\mathbf{z}}_i + \sum_{j \in \hat{\mathcal{N}}(i)} \mathbf{m}_{ij}, \tag{7}$$

GAT [4] involves the normalization of attention values, thus the normalization term is processed separately to obtain the weighted neighbor messages:

$$\begin{aligned} \mathbf{r}_i &= \max\{\mathbf{a}_{ij} \mid j \in \hat{\mathcal{N}}(i)\} \cup \{\bar{\mathbf{u}}_i\}, \\ \gamma(\mathbf{a}_{ij}) &= \exp(\mathbf{a}_{ij} - \mathbf{r}_i), \\ \mathbf{z}_i &= \frac{\beta_i \gamma(\bar{\mathbf{u}}_i)\bar{\mathbf{z}}_i + \sum_{j \in \hat{\mathcal{N}}(i)} \gamma(\bar{\mathbf{a}}_{ij})\mathbf{m}_{ij}}{\beta_i \gamma(\bar{\mathbf{u}}_i) + \sum_{j \in \hat{\mathcal{N}}(i)} \gamma(\bar{\mathbf{a}}_{ij})}, \end{aligned} \tag{8}$$

where $\bar{\mathbf{z}}_i$ is the historical aggregated result for node i, and $\bar{\mathbf{u}}_i$ is the historical maximum attention value among the neighbors of node i, $\mathbf{r}_i$ represents the current maximum attention value among the same set of neighbors. The attention value $\mathbf{a}_{ij}$ between node i and node j is computed using $\mathbf{z}_i$ and $\mathbf{z}_j$ from the previous layer [4]. This approach prevents overflow in the computation of $\mathrm{softmax}(\cdot)$.

Finally, we still need to update the maximum attention value $\mathbf{u}_i$ of node i. The update principle is to maintain the denominator of the corrected $\mathrm{softmax}(\cdot)$ to sum as 1:

$$\mathbf{u}_i = \mathbf{r}_i + \log\left[\beta_i \gamma(\bar{\mathbf{u}}_i) + \sum_{j \in \hat{\mathcal{N}}(i)} \gamma(\mathbf{a}_{ij})\right]. \tag{9}$$

It is crucial to observe that the update is conducted by adding the logarithm of the denominator in the attention formula. Given that the logarithm function can yield both positive and negative outcomes, this ensures that the value of $\mathbf{u}_i$ does not increase indefinitely. Therefore, GAT requires maintaining two historical embeddings: $\bar{\mathbf{z}}_i$ and $\bar{\mathbf{u}}_i$.

Integration with full-batch training: In full-batch training, all labels in the training set must be considered in the loss function computation. With moving message aggregation, only messages within the current target nodes set $\mathcal{V}_b$ are updated, potentially leaving some nodes without the latest input. To address this, we handle self-loop edges for each node individually by using a vertex-cut scheme akin to METIS for graph data partitioning. This allows each node to send messages to itself without distributed communication, ensuring that the self-loop messages from the previous layer for updating current node's state are always the latest and nodes in the GNN are updated at each layer. Unlike previous methods that cached all neighboring historical embeddings in each partition [14], [15], our approach stores historical embeddings on target nodes rather than source nodes. By employing an incremental aggregation technique as shown in (5), we retain aggregated historical embeddings exclusively on the target nodes. The message aggregation computation relies solely on the messages $\mathbf{m}_{ij}$ and the historical embeddings of the target nodes, $\bar{\mathbf{z}}_i$. Moreover, moving message aggregation introduces only a minimal increase in computational complexity due to its reliance on simple element-wise operations. This additional complexity is comparable to that of standard activation functions or dropout layers, which explains its efficiency. By seamlessly integrating these operations, our approach maintains high performance without compromising resource utilization.

Benefits of moving message aggregation: Moving message aggregation strategy implements an incremental approach to message aggregation, reducing redundant computations and storage requirements. Unlike traditional incremental methods, we introduce a flexible moving aggregation form with adaptive dynamic decay coefficients, allowing target nodes to adjust to varying message volumes and use different aggregation functions. To address error accumulation during aggregation, we employ two key approaches. First, the adaptive β_i is derived from variance stability analysis to ensure numerical error remains bounded. Second, decay term $\beta_i \gamma(\bar{\mathbf{u}}_i)$ in (8) filters out outdated information and acts as a stabilizer to prevent near-zero denominators in the softmax function, mitigating numerical instability from diverging values over time. Overall, our moving message aggregation improves upon existing methods by incorporating topology-aware decay and supporting complex aggregators like GAT, effectively reducing error accumulation during training. Furthermore, moving message aggregation introduces only a minimal increase in computational complexity, relying on simple element-wise operations. This added complexity is comparable to standard activation functions or dropout layers, which explains its efficiency. By seamlessly integrating these operations, our approach maintains high performance without compromising resource utilization.

C. Implementation and Analysis

Prototype implementation: By integrating source node-centric chunking with message aggregation, we developed a prototype of the distributed GNN training framework, *Emma*, as detailed in Algorithm 1. The implementation of *Emma* was carried out using PyG [35], utilizing NVIDIA NCCL as the communication backend and employing the $\mathrm{AllToAll}(\cdot)$ mechanism [36] for efficient distributed communication.

Initially, we applied the METIS algorithm [24] to partition the entire graph into P segments, assigning each GPU a corresponding partition. Each partition's data is then randomly divided into B chunks, each containing a set of source nodes. For each chunk $\mathcal{S}_b$, we create a partition-to-partition routing table using $\mathrm{BuildAllToAllRouteTable}(\cdot)$ to facilitate communication across all GNN layers within a single iteration. The current chunk of source nodes $\mathcal{S}_b$ is then broadcast to all partitions, allowing each partition to receive the node messages $\hat{\mathrm{H}}_{\mathcal{S}_{bk}}^{(l-1)}$. These messages are combined with the self-sent messages $\mathrm{H}_{\mathcal{V}_k}^{(l-1)}$ from each node. Moving message aggregation is then performed to update the embeddings $\mathrm{Z}_{\mathcal{V}_k}^{(l)}$, after which the node embeddings $\mathrm{H}_{\mathcal{V}_k}^{(l)}$ for partition k at layer l are retrieved. The

Algorithm 1: Emma Training Framework.

Input: Partition id k, partition graph $\mathcal{G}_k = (\mathcal{V}_k, \mathcal{E}_k, \mathcal{X}_k)$, label $\mathcal{Y}_k$, partition features buffer Z_k, chunk number B, initial model θ_0.
Output: The trained model θ_T.
for $l = 1$ **to** L **do**
  $\bar{\mathrm{Z}}^{(l)}_{\mathcal{V}_k} \leftarrow 0$
end for
$\mathrm{H}^{(0)}_{\mathcal{V}_k} \leftarrow \mathcal{X}_k; t \leftarrow 1$
while $t \leq T$ **do**
  Divide $\mathcal{V}$ into B sets$\{\mathcal{S}_1, \ldots, \mathcal{S}_B\}$ ▷ (3)
  for $b = 1$ **to** B **do**
    $\mathcal{R}_b \leftarrow \mathrm{BuildAllToAllRouteTable}(\mathcal{S}_b)$
    for $l = 1$ **to** L **do**
      $\hat{\mathrm{H}}^{(l-1)}_{\mathcal{S}_{bk}} \leftarrow \mathrm{AllToAll}(\mathrm{H}^{(l-1)}_{\mathcal{V}_k}, \mathcal{R}_b)$
      $\mathrm{Z}^{(l)}_{\mathcal{V}_k} \leftarrow \mathrm{MovingAggr}^{(l)}_{\theta_{t-1}}(\bar{\mathrm{Z}}^{(l)}_{\mathcal{V}_k}, \mathrm{H}^{(l-1)}_{\mathcal{V}_k} \| \hat{\mathrm{H}}^{(l-1)}_{\mathcal{S}_{bk}})$ ▷ (5)
      $\mathrm{H}^{(l)}_{\mathcal{V}_k} \leftarrow \mathrm{Update}^{(l)}_{\theta_{t-1}}(\mathrm{Z}^{(l)}_{\mathcal{V}_k})$
    end for
    $g_t \leftarrow \mathrm{AllReduce}(\frac{\partial \mathcal{L}(\mathrm{H}^{(L)}_{\mathcal{V}_k}, \mathcal{Y}_k)}{\partial \theta_{t-1}})$
    $\theta_t \leftarrow \mathrm{SGD}(\theta_{t-1}, g_t)$
    $t \leftarrow t + 1$
  end for
end while

loss function is computed using the labels $\mathcal{Y}_k$, followed by executing the backpropagation algorithm. Gradients of the node embeddings are propagated to the relevant partitions based on the routing table $\mathcal{R}_b$. Model gradients from each partition are aggregated using the $\mathrm{AllReduce}(\cdot)$ operator, and the optimization algorithm updates the model. This iterative process continues for $T \cdot B$ iterations until convergence. Prior to training, all graph structures and data are loaded into GPU memory to minimize data transfers between the GPU and CPU.

Emma minimizes implementation complexity through three modular components that seamlessly integrate with existing GNN frameworks: (1) *Standard Chunk Loader:* Operates locally within each GPU partition, leveraging native graph partitioning without new communication protocols. (2) *Auto-Grad Enabled Message Router:* Implements a plug-in communication layer for automatic forward message passing and backward gradient synchronization using standard AllToAll primitives. (3) *Drop-in Aggregation Replacement:* Maintains identical input/output interfaces as conventional GNN layers, allowing direct substitution of existing aggregation functions (e.g., pyg.nn.GCNConv) while preserving the original training pipeline. This modular design ensures compatibility with mainstream frameworks and preserves familiar debugging workflows. All critical path operations appear as standard computation graph nodes in profilers like PyTorch Profiler.

Staleness of historical embeddings:Emma achieves better freshness of historical embeddings compared to target-centric sampling methods (TSMs). With the same number of sampled nodes, the source-node-centric chunking strategy samples $|\mathcal{S}_b|$ nodes per iteration and updates $|\mathcal{V}_b| = |\mathcal{V}_{b1} \cup \mathcal{V}_{b2} \cup \cdots \cup \mathcal{V}_{bP}|$, where P is the number of graph partitions and $|\mathcal{V}_b| \geq |\mathcal{S}_b|$. In contrast, traditional TSMs follow the message aggregation path, where the number of updated nodes in the next layer is less than that in the previous layer, i.e., $|\cdot| \leq |\mathcal{S}_b|$.

Furthermore, our method leverages distributed partitioned parallelism during training, allowing cross-partition sharing of node embeddings (activations) in each iteration. Traditional sampling approaches rely on mini-batch training, requiring each partition to redundantly compute and store embeddings. This enables our method to maintain a larger $|\mathcal{S}_b|$, accelerating the propagation of neighbor messages.

Existing historical embedding techniques, such as PyGAS [18] and Sancus [15], introduce some staleness in historical embeddings, despite theoretical assurances. Our source-node-centric chunking and periodic embedding updating guarantee better staleness control than current methods, enhancing convergence properties.

Convergence properties: The moving message aggregation can approximate vanilla message aggregation with various operators at the convergence point. Let $\mathrm{Aggr}(\cdot)$ in (5) be $\mathrm{sum}(\cdot)$. After normalization, neighborhood messages $\mathbf{m}_{ij} \in [-1, 1]$ follow an identical distribution with expectation $\mathbb{E}[\mathbf{m}^*_{ij}]$, and $\mathbf{z}_i$ converges to a stable distribution with expectation $\mathbb{E}[\mathbf{z}^*_i]$. We demonstrate that $\mathbb{E}[\mathbf{m}^*_{ij}]$ and $\mathbb{E}[\mathbf{z}^*_i]$ are linearly related. Given that $\bar{\mathbf{z}}_i$ follows the same distribution as $\mathbf{z}_i$, we can rewrite (5) to obtain $\mathbb{E}[\mathbf{z}^*_i] = \beta_i \mathbb{E}[\mathbf{z}^*_i] + \sum_{j \in \hat{\mathcal{N}}(i)} \mathbf{m}^*_{ij}$. This leads to $\mathbb{E}[\mathbf{z}^*_i] = \frac{\sum_{j \in \hat{\mathcal{N}}(i)} \mathbf{m}^*_{ij}}{1-\beta_i} = \deg_{\text{in}}(i) \frac{\sum_{j \in \hat{\mathcal{N}}(i)} \mathbf{m}^*_{ij}}{|\hat{\mathcal{N}}(i)|}$. We can derive the approximation error using Hoeffding's inequality: For any $\epsilon > 0$,

$$\mathbb{P}\left(\left|\frac{\mathbb{E}[\mathbf{z}^*_i]}{\deg_{\text{in}}(i)} - \mathbb{E}[\mathbf{m}^*_{ij}]\right| < \epsilon\right) \geq 1 - 2\exp\left(-\frac{1}{2}|\hat{\mathcal{N}}(i)|\epsilon^2\right), \tag{10}$$

where the updated node embedding $\mathbb{E}[\mathbf{z}^*_i]$ is bounded by neighborhood messages $\mathbb{E}[\mathbf{m}^*_{ij}]$. The same approximation error bound applies to the $\mathrm{mean}(\cdot)$ and $\mathrm{attention}(\cdot)$ operators depicted in (8).

The optimization process converges with appropriate hyperparameter settings. With $\eta = O(\epsilon^L)$, $\beta_1 = O(\epsilon^L)$, $\beta_2 \in [0, 1]$, $\beta_{v,k} = O(\epsilon^{L-k})$, and $T = O(\epsilon^{-(L+2)})$, *Emma* guarantees convergence to an ϵ-stationary point, ensuring that $\mathbb{E}[\|\nabla F(\mathrm{w}_\tau)\|] \leq \epsilon$ for a randomly selected $\tau \in \{1, \ldots, T\}$. Here, L is the number of GNN layers, $F(\mathrm{w}_\tau)$ denotes the GNN model with weights w after τ training iterations, T is the total number of training iterations, η is the learning rate, and β_1 and β_2 are the exponential decay rates for the Adam optimizer. Additionally, $\beta_{v,k}$ represents the adaptive decay coefficient for node v at layer k, as defined in (6).

The source node-centric chunking aligns with the DropEdge [37] regularization strategy under specific conditions, effectively addressing the over-smoothing issue in GNNs. The following properties hold: (1) The edge selection probability matches that of DropEdge: $\mathbb{P}((j, i) \in \mathcal{E}_b) = \frac{1}{B}$, and (2) Pairwise dependency between edges diminishes asymptotically: $\mathbb{E}|\mathrm{Cov}(\mathbb{I}_{(j,i)}, \mathbb{I}_{(j',i')})| = \mathcal{O}(B^{-1}|\mathcal{V}|^{-1})$ for $(j, i) \neq (j', i')$, where $\mathbb{I}_{(j,i)}$ is the indicator variable for the inclusion of

TABLE II
STATISTICS OF GRAPH DATASETS

	# Nodes	# Edges	# Features	# Classes
Reddit	233K	114M	602	41
Yelp	716K	7M	300	100
Products	2.4M	62M	100	47
Papers100M	111M	1.6B	128	172

TABLE III
HYPERPARAMETER SETTINGS

	# Layers	Hidden units	Dropout p
Reddit	4	256	0.5
Yelp	4	512	0.1
Products	3	256	0.3
Papers100M	3	64	0.5

edge (j, i) in chunk b. These properties ensure that as the number of nodes increases, the source node-centric chunking approaches edge-independent sampling akin to DropEdge regularization, effectively delaying over-smoothing in GNNs while preserving structural stochasticity.

Communication Balancing: Source node-centric chunking ensures balanced communication via two properties: (a) Expectation preservation: $\lim_{|\mathcal{B}_b|\to\infty} \frac{1}{|\mathcal{B}_b|}\sum_{v\in\mathcal{B}_b} D(d_v) \overset{a.s.}{\to} c\cdot\mathbb{E}[d_v]$ for chunk size $|\mathcal{B}_b|$, node load $D(d_v)$, and constant c, which guarantees no systematic communication bias is introduced, and (b) Variance control for L-layer GNN with B chunks: $\mathrm{Var}(\sum_{l=1}^{L} X_b^{(l)}) \le \frac{L}{B}\cdot|\mathcal{V}|^{(3-\gamma)(\gamma-1)}$ where $X_b^{(l)}$ indicates messages inclusion at l layer in chunk b and γ donates power-law exponent. The $\mathcal{O}(L/B)$ growth ensures stable communication without severe fluctuations, improving over traditional methods by $\Theta(|\mathcal{V}|^{(L-1)(3-\gamma)/(\gamma-1)})$.

IV. EVALUATION

A. Experiment Settings

Setups: Our setup consists of six machines interconnected with two-port 100 G NICs. Each machine is equipped with four NVIDIA A40 (48 G), two Xeon 6342R@2.80 GHz, and PCIe4x16 for connecting CPU-GPU and GPU-GPU. We used one of the machines for our primary experiments and also evaluated distributed scalability in Section IV-F.

Graph datasets: We evaluate our method on four commonly used graph datasets, i.e., Reddit [3], Yelp [30], OGB-Products [38], and OGB-Papers100M [39]. Table II lists the statistics of these datasets. For evaluation metrics, we employ micro-f1 score for the multi-label Yelp dataset and classification accuracy for the other single-label datasets.

GNN models: We evaluated our method using two GNN models, GCN and GAT. We used the same hyperparameters for different training methods, as shown in Table III. For the GAT model, we simplified the number of attention heads to 1. We maintained a learning rate of 0.001 and used the Adam optimizer with hyperparameters set to $\beta_1 = 0.9$ and $\beta_2 = 0.999$. Additionally, we incorporated LayerNorm [40] for normalization at each layer.

Baselines: We compared our method with nine state-of-the-art distributed GNN training methods. (1) BNS-GCN [28] samples boundary nodes across partitions and transmits only the sampled nodes to reduce communication. (2) PipeGCN [14] overlaps partition data synchronization with asynchronous communication using historical embeddings for graph convolution. (3) Sancus [15] leverages historical embeddings and synchronizes only the partitions with high staleness to reduce communication overhead. We terminate the partitioned broadcast communication every 2 epochs after the first 50 epochs. (4) GAS [18] combines first-order neighborhood sampling with historical embeddings to approximate full-graph GNN training. (5) LMC [19] optimizes the forward and backward propagation processes based on GAS to ensure convergence. (6) DistDGL [21] accelerates training using a subgraph sampling, implemented using DGL's native distributed parallel framework. (7) AdaQP [41] employs adaptive message quantization to alleviate communication overhead in distributed full-batch training. (8) Sanqus [42] extends Sancus by combining staleness-aware synchronization with message quantization for enhanced training efficiency. (9) NeutronTP [43] introduces tensor parallelism by partitioning data along feature dimensions rather than graph structures to achieve perfect load balancing. To align the hyperparameters across different methods as closely as possible, we set the message sampling probability for BNS-GCN to 0.1 and the chunk size for our method to $B = 10$, ensuring similar communication volumes. For the other methods, which do not have parameters that directly influence communication volume, we use the default settings as specified in their respective papers. For full-batch frameworks, we maintain consistency between the number of partitions and GPUs ($= 4$). For subgraph sampling frameworks like DistDGL, we employ an equivalent GPU configuration ($= 4$). For single-GPU frameworks (LMC and GAS), we conduct experiments using a single GPU. The experimental comparisons are conducted using the open-source code provided by the authors. Most of the codebases offer GCN training components, so our primary comparisons are conducted on the GCN model. Where applicable, we also compare the GAT model using the codebases that support it.

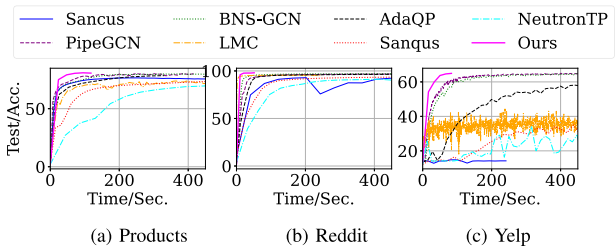
B. Training Efficiency

1) GCN Training Efficiency.: We first evaluate on the GCN model with simple linear message aggregation.

Total time cost: We measured the total time for GCN model training to converge and compared our *Emma* with BNS-GCN, PipeGCN, Sancus, LMC, DistDGL, AdaQP, Sanqus and NeutronTP. The results are presented in Table IV. For the Products, Reddit, and Yelp datasets, we used a single machine with four GPUs for evaluation. For the Papers100 M dataset, we used six machines with a total of 24 GPUs. GAS and LMC are single-GPU frameworks by chunk-wise offloading, while

TABLE IV
TOTAL TIME COST OF GCN TRAINING

Dataset	Time cost (seconds)				
	BNS-GCN	PipeGCN	Sancus	GAS	LMC
Products	519.0	343.3	454.1	2814.3	3152.1
Reddit	837.6	1214.1	4293.0	491.1	542.5
Yelp	1050.5	721.9	—	1396.8	1100.2
Papers100M	2067.1	OOM	OOM	—	—
Dataset	**Time cost (seconds)**				
	DistDGL	AdaQP	Sanqus	NeutronTP	Ours
Products	6002.3	330.8	1063.9	3479.8	125.6
Reddit	1577.6	555.9	4307.7	1280.3	442.3
Yelp	402.2	570.6	1818.2	907.7	395.3
Papers100M	4367.5	OOM	OOM	OOM	985.5



(a) Products (b) Reddit (c) Yelp

Fig. 5. Accuracy curve of GCN training.

DistDGL is a distributed mini-batch training framework using subgraph sampling. The other frameworks train GNN on full-batch mode. The missing value for Sancus on the Yelp dataset is attributed to its failure to converge, as shown in Fig. 5. Notably, our *Emma* achieved the lowest time costs in all testing cases. Specifically, on the Products dataset, *Emma* was 2.73× faster than its best competitor PipeGCN. Overall, *Emma* achieved an average speedup of 2.89×, 2.44×, 6.66×, 9.70×, 17.46×, 1.78×, 7.60× and 10.97× over BNS-GCN, PipeGCN, Sancus, LMC, DistDGL, AdaQP, Sanqus and NeutronTP, respectively, demonstrating its efficiency for graph learning tasks. Our efficiency stems primarily from the lower communication cost of push sampling and the decreased computation cost of moving message aggregation. For the Papers100 M dataset, we only tested the frameworks capable of distributed scaling. It is evident that the available resources were insufficient to train the GCN model in full-batch mode for most frameworks. PipeGCN requires additional buffers for asynchronous communication; Sancus and Sanqus' historical embedding storage reduces their memory scalability; AdaQP's long message processing chain introduces significant storage pressure; and NeutronTP, while reducing intermediate layer memory usage through tensor parallelism, does not reduce the storage overhead for graph structures and demands higher GPU interconnect bandwidth. The DistDGL framework trains only on labeled nodes. Since only about 1.5% of the nodes in the Papers100 M dataset have labels, the amount of data DistDGL trains on is even smaller than that of the Products dataset. This allows DistDGL to complete training in a shorter time. We further discuss the reduction in communication and computation costs in Section IV-E.

Training throughput: We also measure the throughput of training epochs by comparing our *Emma* with BNS-GCN, PipeGCN, Sancus, LMC, DistDGL, AdaQP, Sanqus and NeutronTP, and the results are shown in Table V. Our *Emma* consistently demonstrates the highest throughput for training each epoch on the GCN model across all testing cases. For instance, the top-performing competitor PipeGCN achieves an average of 2.99 epochs per second on the Yelp dataset, while our method achieves an average of 6.83 epochs per second, representing a 2.28× faster training throughput. DistDGL exhibits extremely low training throughput due to the high cost of its subgraph sampling and the redundancy of many nodes being sampled and computed multiple times. Similarly, due to the limited number of labeled nodes in the Papers100 M dataset, its throughput exceeds that of the medium-sized Products dataset. The dense edges in the Reddit dataset increase the cost of subgraph sampling, thereby reducing the training throughput.

TABLE V
THROUGHPUT OF GCN TRAINING

Dataset	Throughput (# epochs/s)				
	BNS-GCN	PipeGCN	Sancus	GAS	LMC
Products	0.85	1.19	1.10	0.21	0.14
Reddit	2.65	2.30	0.25	2.14	1.63
Yelp	2.24	2.99	0.62	1.23	1.09
Paper100M	0.31	OOM	OOM	—	—
Dataset	**Throughput (# epochs/s)**				
	DistDGL	AdaQP	Sanqus	NeutronTP	Ours
Products	1.4×10^{-3}	0.7558	0.6578	0.2298	3.98
Reddit	1.5×10^{-3}	0.8994	0.4016	0.3124	6.33
Yelp	2.6×10^{-2}	0.7010	0.5494	0.7602	6.83
Papers100M	2.3×10^{-2}	OOM	OOM	OOM	0.51

Convergence efficiency: We further compare the convergence efficiency by plotting the accuracy curves for Sancus, PipeGCN, BNS-GCN, LMC, AdaQP, Sanqus, NeutronTP and *Emma* on the GCN model in Fig. 5. The x-axis represents the elapsed training time, and the y-axis indicates the corresponding training accuracy. We can see that *Emma* always achieves the highest training accuracy under the same time overhead, and achieves convergence quickly and stably on all three datasets. On the Yelp dataset, the Sancus method fails to converge. Methods such as LMC, Sanqus, and NeutronTP exhibit under-convergence, which is more pronounced in the Yelp multi-label classification task, making these frameworks more susceptible to their design limitations.

2) GAT Training Efficiency.: We proceed to assess on the GAT model with self-attention based message aggregation. We compare *Emma* with PipeGCN, BNS-GCN, and DistDGL. While these three methods are primarily designed for GCN model training, they can also be extended to accommodate GAT model. We use the GAT implementation provided in the official code of BNS-GCN and integrate it into the official code of PipeGCN.

Total time cost: We measure the total time cost for GAT model training to converge and report the results in Table VI through an early stopping termination strategy. We observe that *Emma* is the fastest method in most cases, except when compared with PipeGCN on Reddit. However, PipeGCN exhibits under-convergence and terminates early on this dataset, while *Emma*'s accuracy continues to improve throughout training, ultimately achieving state-of-the-art (SOTA) performance,

TABLE VI
TOTAL TIME COST OF GAT TRAINING

Dataset	Time cost (seconds)			
	BNS-GCN	PipeGCN	DistDGL	Ours
Products	953.8	336.8	6122.1	179.2
Reddit	3870.5	142.1	1805.6	605.3
Yelp	2317.6	2117.9	708.1	418.3

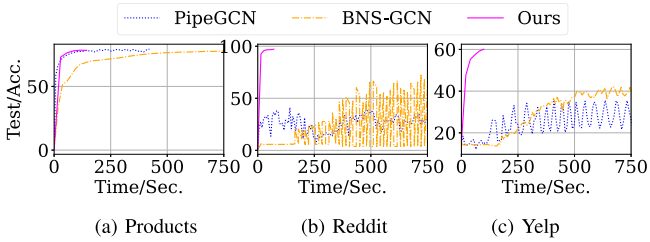
TABLE VII
THROUGHPUT OF GAT TRAINING

Dataset	Throughput (# epochs/s)			
	BNS-GCN	PipeGCN	DistDGL	Ours
Products	0.51	1.18	1.3×10^{-3}	2.79
Reddit	0.66	1.54	1.3×10^{-3}	4.13
Yelp	1.27	1.24	2.2×10^{-2}	5.02

TABLE VIII
ACCURACY OF GCN TRAINING

Dataset	Accuracy (%)				
	BNS-GCN	PipeGCN	Sancus	GAS	LMC
Products	79.50	78.69	75.60	75.63	74.25
Reddit	97.12	97.06	94.40	95.40	95.43
Yelp	65.26	65.28	—	44.23	44.37
Papers100M	61.45	OOM	OOM	—	—

Dataset	Accuracy (%)				
	DistDGL	AdaQP	Sanqus	NeutronTP	Ours
Products	77.87	78.43	74.98	75.08	79.53
Reddit	95.29	96.48	94.35	94.00	96.91
Yelp	64.23	59.86	41.40	40.63	65.21
Papers100M	61.32	OOM	OOM	OOM	61.84



(a) Products (b) Reddit (c) Yelp

Fig. 6. Accuracy curve of GAT training.

TABLE IX
ACCURACY OF GAT TRAINING

Dataset	Accuracy (%)			
	BNS-GCN	PipeGCN	DistDGL	Ours
Products	78.08	77.93	77.79	78.94
Reddit	92.88	41.03	95.38	96.94
Yelp	43.66	39.41	61.48	65.15

as show in Fig. 6 and Table IX On average, *Emma* achieved a speedup of 5.75× over BNS-GCN, 3.47× over PipeGCN, and 12.95× over DistDGL. Notably, our performance improvement on GAT models is more significant than on GCN models, attributed to our incremental aggregation strategy that reduces more computation costs for the comprehensive self-attention-based message aggregation.

Training throughput: We also compare the GAT training throughput and show the results in Table VII. We can see that our model achieves the highest throughput on all datasets, especially on Yelp, where it is 3.95× faster than BNS-GCN and 4.05× faster than PipeGCN. On Products and Reddit, our model is also significantly faster than BNS-GCN, and more than twice as fast as PipeGCN. This demonstrates the efficiency of our incremental message aggregation method for graph attention networks.

Convergence efficiency: We also plot the convergence curves of BNS-GCN, PipeGCN and *Emma* on the GAT model for different datasets in Fig. 6. On the Products datasets, the three methods show similar convergence efficiency. However, on Reddit and Yelp, BNS-GCN and PipeGCN exhibit periodic fluctuations in test accuracy even after several hundreds of epochs, indicating unstable training results and low convergence efficiency. The poor performance of PipeGCN in training GAT models is due to its reliance on stale historical embeddings for each node, which works well with simple GCN aggregation layers but can lead to model instability in complex weighted attention layers. BNS-GCN, while employing a similar message sampling strategy, lacks the design to reduce aggregation variance using historical embeddings, resulting in slow convergence and significant performance fluctuations. In contrast, our proposed method converges to the stable level much faster.

C. GNN Training Accuracy

1) GCN Training Accuracy: We compared the training accuracy of our method with nine baseline models, including BNS-GCN, PipeGCN, Sancus, GAS, LMC, DistDGL, AdaQP, Sanqus and NeutronTP. The test accuracy of the model with the best validation accuracy is shown in Table VIII, following the protocol of previous works [28]. Our model demonstrates comparable accuracy to the best competitor, BNS-GCN, with differences ranging from 0.05% to 0.33%. It is worth noting that BNS-GCN takes approximately 2.98× longer than our model to achieve this accuracy. This highlights the effectiveness and robustness of our model for graph learning tasks.

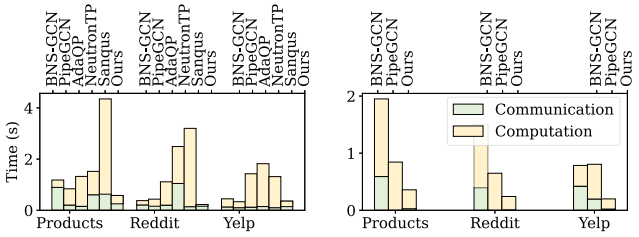
2) GAT Training Accuracy: We compared the accuracy of PipeGCN, BNS-GCN, DistDGL and our proposed *Emma* on the GAT model, with results shown in Table IX. Our method consistently achieved the highest accuracy in GAT training on Reddit and Yelp. In particular, we improve the accuracy on Yelp from 61.65% to 65.15%. This demonstrates the effectiveness and robustness of *Emma* for the more complicated GAT model.

D. Ablation Study

We further conduct ablation studies to analyze the contributions of the proposed methods in terms of accuracy, efficiency, and memory usage, as shown in Table X. Under our adaptive decay coefficient, if the source node-centric chunking is not used and only moving aggregation is applied, the computational form is consistent with the case where neither mechanism is enabled. In terms of throughput, the additional overhead of moving aggregation is negligible. Chunking reduces peak memory usage, while the historical embeddings used in moving aggregation increase memory usage. However, the memory reduction

TABLE X
ABLATION STUDY

Configuration	Products		Reddit		Yelp	
	GCN	GAT	GCN	GAT	GCN	GAT
Accuracy (%)						
w. chunking, w. moving	79.53	78.94	96.91	96.94	65.21	65.15
w. chunking, w.o. moving	78.57	78.62	96.73	93.85	64.36	59.03
w.o. chunking, w.o. moving	79.80	79.24	96.88	96.14	64.16	60.50
Throughput (epoch/s)						
w. chunking, w. moving	4.34	3.41	10.05	7.13	6.12	5.05
w. chunking, w.o. moving	4.33	3.41	10.05	7.12	6.12	5.04
w.o. chunking, w. moving	1.26	0.82	1.72	1.00	3.13	2.01
w.o. chunking, w.o. moving	1.28	0.83	1.73	1.01	3.18	2.06
Peak Memory (MB)						
w. chunking, w. moving	9834.63	9093.02	2162.45	2080.74	5489.39	5841.62
w. chunking, w.o. moving	8408.85	7770.40	1856.08	1889.73	4933.10	5134.37
w.o. chunking, w. moving	12195.00	14447.21	2831.57	5241.87	5994.68	6936.98
w.o. chunking, w.o. moving	10768.42	13129.20	2526.72	5059.35	5439.34	6234.22



(a) GCN model (b) GAT model

Fig. 7. Time costs breakdown.

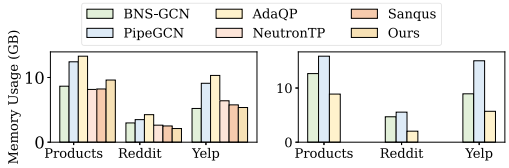
achieved by chunking is significantly greater, effectively offsetting the additional memory overhead introduced by moving aggregation.

E. Breakdown Analysis

1) Computation and Communication Costs: To analyze the performance improvements, we decompose the average training time of each epoch into communication and computation time. PipeGCN overlaps computation and communication for parallel acceleration, and we define the non-overlapping communication cost as the communication time. Fig. 7(a) and (b) present the results for the GCN and GAT models, respectively.

We observe that the GAT model requires significantly more computation time than the GCN model due to its self-attention-based aggregation mechanism. Notably, for the GAT model, PipeGCN incurs no communication time on Products and Reddit, indicating that communication is entirely integrated within the computation. Overall, our method consistently achieves the lowest communication cost by transferring only relevant messages to the sampled node in the current epoch, thereby avoiding the message passing costs from unsampled nodes. In terms of computation time, our approach either outperforms or matches BNS-GCN, which discards all messages from unsampled nodes. We leverage buffered historical aggregation values to approximate the untransferred missing messages, thus enhancing training accuracy. In contrast, PipeGCN aggregates messages from all neighbors without sampling, resulting in inferior performance compared to our method.

While Sanqus reduces communication overhead through adaptive quantization, it incurs additional computational costs, leading to decreased overall performance. NeutronTP achieves



(a) GCN model (b) GAT model

Fig. 8. Peak memory usage during GNN training.

effective communication load balancing via tensor parallelism but converts sparse communication between graph partitions into dense communication between tensors, which increases demands on network bandwidth.

2) Peak Memory Usage: We analyze the memory usage of *Emma* and compare it with PipeGCN and BNS-GCN. We measured the average peak memory usage per GPU during GCN and GAT training, with results shown in Fig. 8(a) and (b), respectively. For GCN models, our method demonstrated lower memory usage than PipeGCN and slightly higher usage than BNS-GCN. This is because both BNS-GCN and our method construct a subgraph from the original graph before each epoch, contributing to efficient memory utilization.

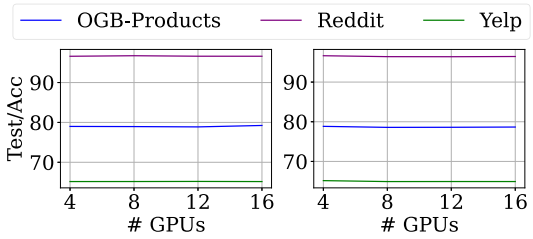
Our approach introduces a buffer to store historical aggregation values, which adds some storage overhead. However, the buffer size is proportional to the number of target nodes and is comparable to the space required for a single element-wise computation. With optimizations in the model code, *Emma* achieves memory efficiency that matches or even surpasses that of BNS-GCN. In contrast, PipeGCN's asynchronous communication requires each partition to allocate send and receive buffers independently of the computation graph, resulting in significantly higher memory consumption. Additionally, for GAT models with complex message passing, *Emma* aggregates fewer messages, which reduces the need for temporary storage and leads to significantly lower memory usage.

F. Scalability on Multiple GPUs and Machines

We evaluate the scalability of our method across different cluster configurations. Specifically, we train *Emma* using varying numbers of GPU servers, each equipped with 4 GPUs interconnected via a 100 G×2 network. We utilize the METIS [24] graph partitioning algorithm for graph partitioning during distributed training.

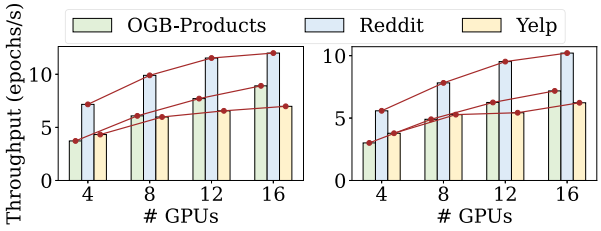
1) Distributed Training Accuracy: We assess the convergence accuracy for both GCN and GAT models, presenting the results in Fig. 9. Our analysis reveals that as the number of GPUs increases, the training accuracy remains stable for both models. This demonstrates the reliable performance of *Emma* in distributed training across multiple GPUs and machines.

2) Distributed Training Throughput: We evaluate the training throughput of both the GCN and GAT models, with results illustrated in Fig. 10. Our analysis shows that as the number of GPUs increases from 4 to 16, the training throughput consistently improves for both models, highlighting the strong scalability of our approach. However, we observe slightly diminished scalability on the Yelp dataset when transitioning from 8 to 12



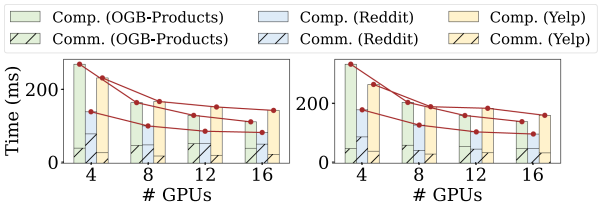
(a) GCN model (b) GAT model

Fig. 9. Accuracy in distributed training.



(a) GCN model (b) GAT model

Fig. 10. GNN training throughput in distributed training.



(a) GCN model (b) GAT model

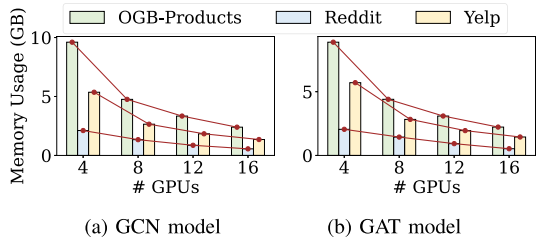
Fig. 11. Time cost breakdown in distributed training.

GPUs. This is due to the graph's sparse structure and higher representation dimensions, which increase the computational load for dense matrix projection operations. Consequently, the rising communication costs associated with more partitions offset the computational efficiency gains from distributed scaling.

3) Time Cost Breakdown: We conducted a detailed analysis of the time breakdown for the GCN and GAT models using various numbers of GPUs, as shown in Fig. 11. Generally, computation time decreases with an increasing number of partitions since each partition processes a smaller portion of the graph. However, we observe minor fluctuations in communication time due to the need for point-to-point communication among all partitions over a bandwidth-limited network. Additionally, the overall volume of network communication increases with more partitions, leading to reduced communication scalability.

4) Per GPU Memory Usage: We also measured the average peak memory usage per GPU for the GCN and GAT models in Fig. 12. As the number of GPUs increases, per GPU memory usage shows a decreasing trend, as training data can be effectively distributed across multiple GPUs.

5) Communication Load Balancing: We evaluated the communication load balancing factor for node-wise subgraph sampling [3], layer-wise subgraph sampling [13], and our proposed



(a) GCN model (b) GAT model

Fig. 12. Peak memory usage in distributed training.

TABLE XI
AVERAGE COMMUNICATION IMBALANCE FACTOR IN DISTRIBUTED SAMPLING ON REDDIT (LOWER IS BETTER)

# Partitions.	2	4	6	8	10	12	14	16
Node-wise subgraph sampling	1.01	1.44	1.64	1.72	3.02	3.09	3.04	3.39
Layer-wise subgraph sampling	1.21	1.47	1.60	1.72	2.68	2.76	2.96	3.04
Source node-centric chunking	1.04	1.09	1.15	1.21	1.26	1.30	1.34	1.38

source node-centric chunking on the Reddit dataset. The balancing factor, defined as the ratio of the maximum to the minimum number of messages sent by partitions in a single communication cycle, indicates better balance when closer to 1. We used a unified batch/chunk size of 8192 and a 3-layer GNN model. In node-wise sampling, each node samples 8 neighbors, while layer-wise sampling samples a number of nodes equal to the batch/chunk size. We recorded the average imbalance factors during training. Results in Table XI show that our source node-centric chunking achieves superior balance when the number of partitions is four or more. This improvement is due to the uniform chunking of source nodes in our approach, while target node-centric sampling lacks source node constraints for load balancing.

G. Parameter Configurations

Our proposed method, *Emma*, allows us to adjust the number of chunks B in a single sampling iteration to achieve different performance levels. This is analogous to the sample rate in BNS-GCN [28] or the drop probability in DropEdge [37]. For example, setting $B = 5$ corresponds to a sample rate of $1/5 = 20\%$ for dropping messages at graph boundaries.

The moving aggregation strategy we employ prioritizes the most recent aggregated messages during training. However, as the number of chunks increases, the weight of the latest aggregation results diminishes, making it more challenging for the model to approximate the most current global neighbor aggregation. Conversely, this approach results in smoother moving aggregation operations. We evaluate the impact of different chunk settings on various performance aspects.

1) Impact on Training Throughput: We first evaluate the effect of different settings of B on training throughput. The results for the GCN and GAT models are presented in Fig. 13(a) and (b), respectively. For both models, training throughput increases with larger B. On the Reddit dataset, the benefits are particularly pronounced due to its relatively dense structure, which features a high number of messages and edges. Increasing B significantly reduces message communication and aggregation computation, resulting in more substantial performance gains.

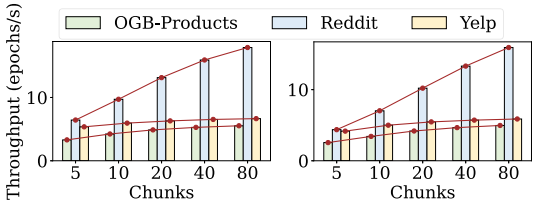


Fig. 13. Throughput under different number of chunks.

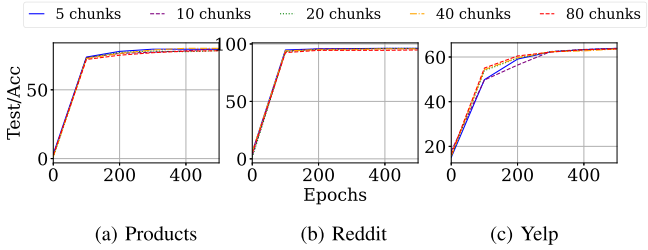


Fig. 14. Accuracy curve for GCN models with different chunks.

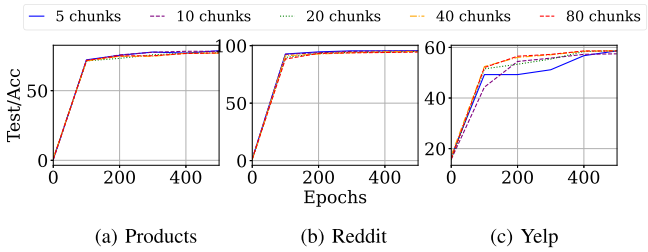


Fig. 15. Accuracy curve for GAT models with different chunks.

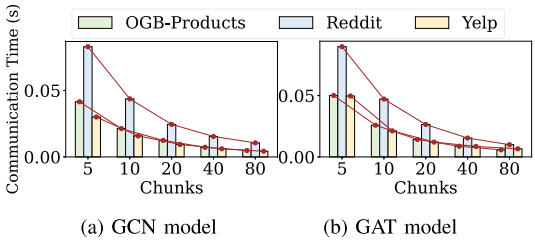


Fig. 16. Communication time with different chunks.

2) Impact on Convergence Efficiency: We also investigate how model accuracy varies with different settings of B. By fixing other parameters and adjusting the number of chunks, we observe corresponding changes in accuracy and performance, as shown in Figs. 14 and 15. As the number of chunks increases, model convergence becomes more stable while causing negligible accuracy drops (<0.5%) on some datasets. According to (5), a larger B results in higher average values of β_i for each node, leading to smoother updates in the moving aggregation results.

3) Impact on Communication Time: We examined the changes in communication time of our method with different chunk settings. The experimental results are shown in Fig. 16. As the number of chunks increases, communication time decreases significantly. This reduction occurs because, for a given number of chunks B, only $\frac{1}{B}$ of the nodes and edges in the graph participate in communication per epoch on average.

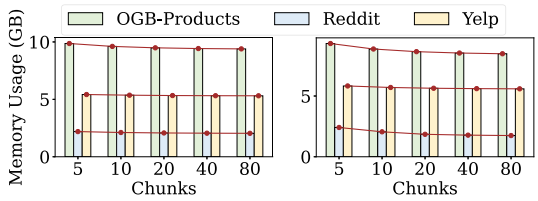


Fig. 17. Peak memory usage with different chunks.

4) Impact on Memory Usage: We also investigated how peak memory usage varies with different numbers of chunks. The variation in memory utilization for the GCN model is illustrated in Fig. 17, where the horizontal axis represents the number of chunks and the vertical axis shows memory usage in gigabytes (GB). The results indicate that the number of chunks has no significant impact on peak memory usage during the training process of the GCN model.

V. Related Works

Distributed GNN Training: Several methodologies complement the discussions in Section II-A regarding distributed GNN training [14], [15], [22], [26], [27], [28], [29], [44], [45], [46], [47]. To reduce communication overhead in partition-parallel training, DIGEST [45] uses a centralized parameter server to synchronize node embeddings, avoiding peer-to-peer communication. AdaQP [46] enhances data transmission via adaptive quantization. Sanqus [42] reduces communication and storage pressure by dynamically sensing the staleness of embeddings and quantizing partition messages. NeutronTP [43] introduces tensor parallelism in GNN training, achieving complete communication load balancing and minimizing the storage of intermediate layers. In contrast, DSP [47] employs a parallel pipeline with a distributed sampler to boost GPU utilization. P3 [48] partitions graph node features and executes the initial GNN layer within each partition, reducing distributed feature retrieval costs. NeutronStar [49] leverages hybrid dependency management for efficient communication by accessing dependent data from neighboring partitions or local caches.

Graph Learning Frameworks: Most open-source frameworks for graph learning support graph neural networks based on the message-passing paradigm [35], [50]. The most widely used frameworks are PyTorch Geometric [35] (PyG) and Deep Graph Library [50] (DGL). PyG adopts a data-centric approach, utilizing Gather and Scatter operators to efficiently propagate messages between nodes and edges. In contrast, DGL allows users to define message-passing computations directly from a graph-centric perspective. Both frameworks continuously evolve with new features and models. Numerous graph neural network frameworks tailored to specific graph data types extend their capabilities based on PyG and DGL foundations, including PyG-Temporal [51], DistDGL [21], and DistTGL [52].

VI. Conclusion

This paper presents *Emma*, a distributed GNN training framework that enhances the efficiency and accuracy of existing

historical embedding-based sampling methods. *Emma* employs a source node-centric chunking technique to mitigate the staleness of historical embeddings while improving communication balance and efficiency. Additionally, it utilizes a moving message aggregation strategy to minimize redundant aggregation of historical embeddings, significantly reducing computation and storage costs. Extensive evaluations demonstrate the effectiveness of *Emma* in terms of training efficiency, accuracy, memory usage, and scalability across various models and datasets.

REFERENCES

[1] 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, Jan. 2022.

[2] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *Proc. Int. Conf. Learn. Representations*, 2017, pp. 1–14.

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

[4] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *Proc. Int. Conf. Learn. Representations*, 2018, pp. 1–12.

[5] J. Lee, I. Lee, and J. Kang, "Self-attention graph pooling," in *Proc. Int. Conf. Mach. Learn.*, 2019, pp. 3734–3743.

[6] M. Zhang, Z. Cui, M. Neumann, and Y. Chen, "An end-to-end deep learning architecture for graph classification," in *Proc. AAAI Conf. Artif. Intell.*, 2018, Art. no. 544.

[7] K. Chen et al., "Distribution knowledge embedding for graph pooling," *IEEE Trans. Knowl. Data Eng.*, vol. 35, no. 8, pp. 7898–7908, Aug. 2023.

[8] W. Yu, X. Lin, J. Liu, J. Ge, W. Ou, and Z. Qin, "Self-propagation graph neural network for recommendation," *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 12, pp. 5993–6002, Dec. 2022.

[9] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2018, pp. 974–983.

[10] X. Wang, X. He, Y. Cao, M. Liu, and T.-S. Chua, "KGAT: Knowledge graph attention network for recommendation," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2019, pp. 950–958.

[11] Q. Guo et al., "A survey on knowledge graph-based recommender systems," *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 8, pp. 3549–3568, Aug. 2022.

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

[13] 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. Int. Conf. Neural Inf. Process. Syst.*, 2019, Art. no. 1009.

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

[15] J. Peng, Z. Chen, Y. Shao, Y. Shen, L. Chen, and J. Cao, "Sancus: Staleness-aware communication-avoiding full-graph decentralized training in large-scale graph neural networks," *Proc. VLDB Endowment*, vol. 15, pp. 1937–1950, 2022.

[16] W. Cong, R. Forsati, M. Kandemir, and M. Mahdavi, "Minimal variance sampling with provable guarantees for fast training of graph neural networks," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2020, pp. 1393–1403.

[17] J. Chen, J. Zhu, and L. Song, "Stochastic training of graph convolutional networks with variance reduction," in *Proc. Int. Conf. Learn. Representations*, 2018, pp. 1–20.

[18] M. Fey, J. E. Lenssen, F. Weichert, and J. Leskovec, "GNNAutoScale: Scalable and expressive graph neural networks via historical embeddings," in *Proc. Int. Conf. Mach. Learn.*, 2021, pp. 3294–3304.

[19] Z. Shi, X. Liang, and J. Wang, "LMC: Fast training of GNNs via subgraph sampling with provable convergence," in *Proc. Int. Conf. Learn. Representations*, 2023, pp. 1–32.

[20] H. Yu, L. Wang, B. Wang, M. Liu, T. Yang, and S. Ji, "GraphFM: Improving large-scale GNN training via feature momentum," in *Proc. Int. Conf. Mach. Learn.*, 2022, pp. 25684–25701.

[21] D. Zheng et al., "DistDGL: Distributed graph neural network training for billion-scale graphs," in *Proc. IEEE/ACM 10th Workshop Irregular Appl.: Architect. Algorithms*, 2020, pp. 36–44.

[22] H. Mostafa, "Sequential aggregation and rematerialization: Distributed full-batch training of graph neural networks on large graphs," in *Proc. Int. Conf. Mach. Learn. Syst.*, 2022, pp. 265–275.

[23] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *Proc. Int. Conf. Mach. Learn.*, 2017, pp. 1263–1272.

[24] G. Karypis and V. Kumar, "METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices," Dept. Comput. Sci., Univ. Minnesota, Minneapolis, MN, USA, Tech. Rep. 97-061, 1997.

[25] N. Xu, B. Cui, L. Chen, Z. Huang, and Y. Shao, "Heterogeneous environment aware streaming graph partitioning," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 6, pp. 1560–1572, Jun. 2015.

[26] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken, "Improving the accuracy, scalability, and performance of graph neural networks with ROC," in *Proc. Int. Conf. Mach. Learn. Syst.*, 2020, pp. 187–198.

[27] L. Ma et al., "Neugraph: Parallel deep neural network computation on large graphs," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2019, pp. 443–457.

[28] C. Wan, Y. Li, A. Li, N. S. Kim, and Y. Lin, "BNS-GCN: Efficient full-graph training of graph convolutional networks with partition-parallelism and random boundary node sampling," in *Proc. Int. Conf. Mach. Learn. Syst.*, 2022, pp. 673–693.

[29] A. Tripathy, K. Yelick, and A. Buluç, "Reducing communication in graph neural network training," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2020, Art. no. 70.

[30] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "Graph-SAINT: Graph sampling based inductive learning method," in *Proc. Int. Conf. Learn. Representations*, 2020, pp. 1–19.

[31] J. Dean et al., "Large scale distributed deep networks," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2012, pp. 1223–1231.

[32] M. Li et al., "Scaling distributed machine learning with the parameter server," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2014, pp. 583–598.

[33] S. Li et al., "PyTorch distributed: Experiences on accelerating data parallel training," *Proc. VLDB Endowment*, vol. 13, pp. 3005–3018, 2020.

[34] J. Chen, J. Zhu, and L. Song, "Stochastic training of graph convolutional networks with variance reduction," in *Proc. Int. Conf. Mach. Learn.*, 2018, pp. 941–949.

[35] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *Proc. Int. Conf. Learn. Representations*, 2019, pp. 1–9.

[36] PyTorch, "Distributed communication package," 2024. [Online]. Available: https://pytorch.org/docs/stable/distributed.html

[37] Y. Rong, W. Huang, T. Xu, and J. Huang, "DropEdge: Towards deep graph convolutional networks on node classification," in *Proc. Int. Conf. Learn. Representations*, 2020, pp. 1–17.

[38] W. Hu et al., "Open graph benchmark: Datasets for machine learning on graphs," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2020, Art. no. 1855.

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

[40] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," 2016, *arXiv:1607.06450*.

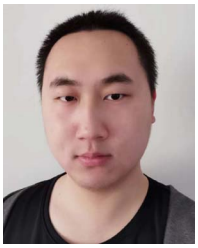
[41] B. Wan, J. Zhao, and C. Wu, "Adaptive message quantization and parallelization for distributed full-graph GNN training," in *Proc. Int. Conf. Mach. Learn. Syst.*, 2023, pp. 203–218.

[42] J. Peng et al., "From Sancus to Sancus^q: Staleness- and quantization-aware full-graph decentralized training in graph neural networks," *VLDB J.*, vol. 34, no. 2, 2025, Art. no. 22.

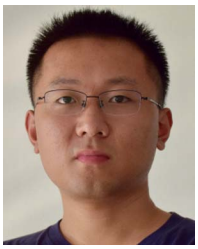
[43] X. Ai et al., "NeutronTP: Load-balanced distributed full-graph GNN training with tensor parallelism," *Proc. VLDB Endowment*, vol. 18, pp. 173–186, 2024.

[44] A. D. Pazho, G. A. Noghre, A. A. Purkayastha, J. Vempati, O. Martin, and H. Tabkhi, "A survey of graph-based deep learning for anomaly detection in distributed systems," *IEEE Trans. Knowl. Data Eng.*, vol. 36, no. 1, pp. 1–20, Jan. 2024.

[45] Z. Chai, G. Bai, L. Zhao, and Y. Cheng, "Distributed graph neural network training with periodic historical embedding synchronization," 2022, *arXiv:2206.00057*.
[46] B. Wan, J. Zhao, and C. Wu, "Adaptive message quantization and parallelization for distributed full-graph GNN training," in *Proc. Int. Conf. Mach. Learn. Syst.*, 2023, pp. 203–218.
[47] Z. Cai et al., "DSP: Efficient GNN training with multiple GPUs," in *Proc. 28th ACM SIGPLAN Annu. Symp. Princ. Pract. Parallel Program.*, 2023, pp. 392–404.
[48] S. Gandhi and A. P. Iyer, "P3: Distributed deep graph learning at scale," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2021, pp. 551–568.
[49] Q. Wang, Y. Zhang, H. Wang, C. Chen, X. Zhang, and G. Yu, "NeutronStar: Distributed GNN training with hybrid dependency management," in *Proc. 2022 Int. Conf. Manage. Data*, 2022, pp. 1301–1315.
[50] M. Wang et al., "Deep graph library: A graph-centric, highly-performant package for graph neural networks," 2019, *arXiv: 1909.01315*.
[51] B. Rozemberczki et al., "PyTorch geometric temporal: Spatiotemporal signal processing with neural machine learning models," in *Proc. ACM Int. Conf. Inf. Knowl. Manage.*, 2021, pp. 4564–4573.
[52] H. Zhou, D. Zheng, X. Song, G. Karypis, and V. Prasanna, "DistTGL: Distributed memory-based temporal graph neural network training," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2023, Art. no. 39.



Wenjie Huang is currently working toward the PhD degree with Zhejiang University, focusing on graph neural networks (GNNs) and their training systems. His research aims to develop efficient algorithms to improve the performance and scalability of graph-based machine learning models, contributing to advancements in artificial intelligence and distributed systems.



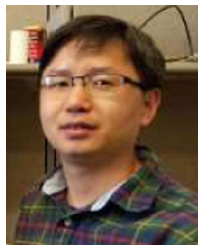
Tongya Zheng received the BEng degree from the Nanjing University of Science and Technology, in 2017, and the PhD degree from the College of Computer Science, Zhejiang University, in 2023. He is with Big Graph Center, School of Computer and Computing Science, Hangzhou City University. His research interests include graph neural networks, temporal graphs, and explanation for artificial intelligence.

Rui Wang received the PhD degree from the University of Science and Technology of China (USTC), in 2021, and now is a ZJU100 research fellow with the School of Software Engineering, Zhejiang University (ZJU). Her research interests lie in graph computing and storage systems, graph learning frameworks, machine learning systems, etc.

Tongtian Zhu is currently working toward the PhD degree with the Computer Science Department, Zhejiang University. His current research focuses on the theoretical foundations of decentralized learning. He pioneers the first theoretical work that connects decentralized SGD with the SAM algorithm. He has served as a reviewer for top-tier venues, including ICLR, NeurIPS, ICML, AISTATS, UAI.

Bingde Hu received the PhD degree in computer science and technology from the College of Computer Science, Zhejiang University. He is a postdoctoral researcher with Zhejiang University. His research interests include graph inference, network embedding, and interpretable machine learning.

Shuibing He (Member, IEEE) received the PhD degree in computer science and technology from the Huazhong University of Science and Technology, in 2009. He is now a ZJU100 Young professor with the College of Computer Science and Technology, Zhejiang University, China. His research areas include intelligent computing, high-performance computing, memory, and storage systems. He is a member of the ACM.



Mingli Song (Senior Member, IEEE) received the PhD degree in computer science from Zhejiang University, China, in 2006. He is currently a professor with the Microsoft Visual Perception Laboratory, Zhejiang University. His research interests include face modeling and facial expression analysis. He received the Microsoft Research Fellowship in 2004.

Xinyu Wang received the graduate and PhD degrees in computer science from Zhejiang University, China, in 2002 and 2007, respectively. He was a research assistant with the Zhejiang University, from 2002 to 2007. He is currently a professor with the College of Computer Science, Zhejiang University. His research interests include streaming data analysis, formal methods, very large information systems, and software engineering.

Sai Wu received the PhD degree from the National University of Singapore (NUS), in 2011, and now is a professor with the College of Computer Science, Zhejiang University. His research interests include distributed database, AI for database, and native AI database system. He has won the best paper awards at VLDB 2014 and SIGMOD 2023. He has served as a Program Committee member for VLDB, ICDE, SIGMOD, and KDD.

Chun Chen is currently a professor with the College of Computer Science, Zhejiang University. His research interests include computer vision, computer graphics, and embedded technology.