PipeQS: Pipeline-Based Adaptive Quantization and Staleness-Aware Distributed GNN Training System

Donghang Wu[1]*, Lian Shen[1]*, Changzhi Jiang[1], Yanhao Li[1], and Xiangrong Liu[1,2,3] ✉

[1]Department of Computer Science and Technology, Xiamen University, Xiamen, China
[2]National Institute for Data Science in Health and Medicine, Xiamen University, Xiamen 361005, China
[3]Xiamen Key Laboratory of Intelligent Storage and Computing
xrliu@xmu.edu.cn

Abstract. Graph Neural Networks (GNNs) have emerged as the state-of-the-art method for graph-based learning tasks. However, training GNNs at scale remains challenging, limiting the exploration of more sophisticated GNN architectures and their application to large real-world graphs. In distributed GNN training, communication overhead and waiting times have become major performance bottlenecks. To address these challenges, we propose PipeQS, an adaptive quantization and staleness-aware pipeline distributed training system for GNNs. PipeQS dynamically adjusts the bit-width of message quantization and manages staleness to reduce both communication overhead and communication waiting time. By detecting pipeline bottlenecks caused by synchronization and utilizing cached communication to bypass message delays, PipeQS significantly improves training efficiency. Experimental results validate the effectiveness of PipeQS, showing up to an 8.3× improvement in throughput while maintaining full-graph accuracy. Furthermore, our theoretical analysis demonstrates fast convergence at a rate of $O(T^{-\frac{1}{2}})$, where T is the total number of training epochs. PipeQS achieves a well-balanced trade-off between training speed and accuracy, significantly reducing training time without compromising performance. The code is available at https://github.com/suupahako/PipeQS-code.

Keywords: Distributed GNN Training · Quantization · Staleness-Aware · Pipeline.

1 Introduction

Graph Neural Networks (GNNs) have become an advanced technique for handling graph-structured data [10], demonstrating exceptional performance in tasks such as node classification [23], link prediction [27], graph classification [10],

* These authors contributed equally to this work.

and recommendation systems [25]. However, training on large-scale graphs remains complex and challenging [10], as the growth in the size of the graph can quickly consume memory and computational resources due to the vast number of node features and the enormous adjacency matrix. This limits the exploration of more complex GNN architectures and practical applications on large real-world graphs.

To address the challenges of training on large-scale graphs, researchers have developed sampling-based methods such as GraphSAGE [5] and VR-GCN [1], which reduce the full graph to mini-batches through neighbor sampling, or by extracting subgraphs as training samples, like Cluster-GCN [2] and GraphSAINT [26]. Although these methods reduce computational resources, they introduce approximation errors and suffer from gradient variance problems, where the randomness in sampling leads to unstable gradient estimates, slowing convergence and reducing model accuracy, especially as graph data scales.

In addition to sampling-based methods, distributed full-graph training has emerged as a promising approach for handling large-scale graph training. This method partitions a large graph into smaller subgraphs, each capable of fitting into a single machine, and communication occurs between these machines to train the partitioned subgraphs. Early works such as NeuGraph [12], ROC [8], CAGNET [18], and Dorylus [17] have demonstrated the significant potential of distributed GNN training. Although distributed full-graph training can retain complete full-graph structural information, it requires frequent information exchange between partition nodes, which leads to a significant increase in communication traffic and seriously affects training efficiency. Additionally, the computation of the local nodes needs to asynchronously receive messages from the remote nodes, which causes unnecessary waiting times during communication phases, further degrading overall throughput. Communication overhead and latency have become the major bottlenecks of distributed full-graph training.

Some studies have explored ways to improve distributed full-graph training to address the above bottlenecks, including message quantization and stale-based methods. For the first bottleneck, BNS-GCN [20] reduces communication time by sampling boundary nodes. AdaQP [19] and EC-Graph [16] apply quantization techniques to compress node features, significantly reducing communication volume. However, the compression error introduced is not conducive to the scalability of these methods. Staleness-based methods have been used to solve the second problem, such as SANCUS [15], which provides a strategy to improve training efficiency by allowing stale updates to replace fresh communication. PipeGCN [21] introduces a pipeline mechanism that uses feature staleness and gradient staleness to reduce communication waiting times. Since stale features are used, these methods require longer training time to ensure the convergence rate of the model. The above methods cannot address both issues at once and often face a trade-off between communication efficiency and convergence stability. Therefore, achieving a balance between minimizing communication overhead and maintaining convergence in distributed full-graph training remains a challenge.

In this work, we propose PipeQS, a novel distributed GNN training method that, for the first time, integrates quantification and staleness within a pipeline framework. To address the potential convergence degradation typically associated with these techniques, we introduce an adaptive adjustment mechanism for bit-width and the number of staleness iterations. Our method effectively reduces communication overhead and solves the communication waiting problem, striking an ideal balance between convergence stability and communication time. PipeQS achieves excellent communication efficiency while maintaining a basic convergence rate of $O(T^{-\frac{1}{2}})$, making it a robust solution for large-scale distributed GNN training.

Our main contributions are summarized as follows:

- We propose PipeQS, a distributed GNN training method that innovatively integrates quantification and staleness within a pipeline framework, addressing the challenges of communication overhead in large-scale GNN training.
- We introduce an adaptive mechanism for adjusting bit-width and the number of staleness iterations, ensuring that the convergence rate is maintained at $O(T^{-\frac{1}{2}})$. This allows the method to achieve strong communication efficiency while preserving the basic convergence rate.
- Our theoretical analysis and empirical evaluations confirm the effectiveness of PipeQS in achieving efficient GNN training while maintaining convergence stability, even as graph size increases. PipeQS achieves significant speedups across datasets, reaching up to 5.06x on Ogbn-products and 4.51x on Reddit.

2 Related Works

2.1 Graph Neural Networks (GNNs)

Graph Neural Networks (GNNs) take graph-structured data as input and aim to learn feature vectors (embeddings) for each node in the graph. At each layer, GNNs perform two core operations: neighbor aggregation and node update, which can be expressed as:

$$z_v^\ell = \phi^\ell(\left\{h_u^{(\ell-1)} | u \in N(v)\right\}), \tag{1}$$

$$h_v = \psi^\ell\left(z_v^\ell, h_v^{(\ell-1)}\right), \tag{2}$$

where $N(v)$ represents the neighbor set of node v, h_v^ℓ is the learned embedding of node v at layer ℓ, z_v^ℓ is an intermediate feature computed through an aggregation function ϕ^ℓ, and ψ^ℓ is the function that updates the node's feature. In this work, our proposed method, PipeQS, builds upon GCN as the baseline model. In the original GCN model [10], the aggregation function ϕ^ℓ uses a weighted average, and the update function ψ^ℓ is defined as a single-layer perceptron, $\sigma(W^\ell(z_v^\ell))$, where $\sigma(\cdot)$ denotes a non-linear activation function, and W^ℓ is the learnable weight matrix.

2.2 Distributed GNN Training

GNNs excel in tasks like node classification and link prediction by aggregating neighboring features to update node representations [10, 23]. However, scaling GNNs to large graphs with millions of nodes poses challenges due to increasing memory and computational demands [7]. Distributed GNN training mitigates this by partitioning large graphs into subgraphs, training them in parallel while exchanging boundary node features. Early works such as NeuGraph [12], AliGraph [24], ROC [8], and Dorylus [17] pioneered this approach but face the persistent challenges of communication overhead and communication waiting. As partitions increase, the number of boundary nodes grows, inflating communication costs. BNS-GCN [20] reduces communication via boundary node sampling but introduces gradient variance, destabilizing training.

2.3 Quantization-based Methods

Quantization is another effective strategy to reduce communication overhead, especially in large-scale graphs. AdaQP [19] dynamically adjusts the bit-width of messages, significantly lowering communication delays. Other techniques, such as SGQuant [3], use hierarchical quantization to minimize memory and bandwidth needs. However, these methods introduce update variance due to quantization errors, impacting convergence stability, especially for large graphs with high-dimensional features.

2.4 Asynchronous Distributed Training Methods

Asynchronous training methods aim to reduce communication waiting times by using stale gradients or features. In deep learning, systems like Hogwild! [13], SSP [6], and MXNet [11] have employed asynchronous updates to hide communication costs. For GNNs, PipeGCN [21] performs parallel communication and computation, utilizing stale features to reduce synchronization delays. However, it assumes a balance between communication and computation time, which may not always hold, limiting its efficiency gains. SANCUS [15] and EC-Graph [16] similarly reduces communication by allowing stale updates, caching embeddings, and skipping communication when possible. While effective in reducing communication, balancing communication efficiency and convergence stability remains a challenge in such staleness-based methods.

In contrast, PipeQS integrates quantization and staleness into a unified framework. It dynamically adjusts bit-width and staleness to achieve a convergence rate of $O(T^{-1/2})$, significantly reducing communication overhead and maintaining high training efficiency and model accuracy across various graph sizes and complexities.

3 The Proposed Framework

Overview In this section, We propose a novel strategy, PipeQS. PipeQS utilizes a pipeline approach that parallelizes communication and computation as much as possible. It applies quantization to features and gradients to reduce the communication costs associated with boundary nodes. Furthermore, PipeQS incorporates features/gradients staleness, allowing the training process to use stale external node features or gradients when communication is incomplete, allowing that computation continues without waiting. This significantly reduces the overall time required for convergence.

To the best of our knowledge, there has been limited research that integrates quantization in pipeline training for distributed GNNs, and combining both quantization and staleness in distributed GNN training remains relatively unexplored. As a result, proving the convergence of such an approach presents a unique challenge. This work aims to theoretically and empirically demonstrate the convergence of this novel pipeline GNN training method that leverages both quantization and staleness. Additionally, we show that the convergence speed is comparable to GNN training methods that solely rely on staleness. We further propose an adaptive bit-width and staleness adjustment mechanism, providing convergence guarantees for PipeQS.

3.1 Challenges in Partitioned Parallel Training

Table 1. Vanilla method communication overhead

Dataset	Partitions	Comm. Ratio
Reddit	2	94.85%
	4	79.47%
Yelp	2	49.42%
	4	69.82%
Ogbn-products	4	68.53%
	8	76.79%

Large Communication Overhead In partition-parallel training, each partition contains local inner nodes and boundary nodes from other partitions, which are critical for neighbor aggregation in GNNs. Updating a node's features requires information from neighboring nodes across partitions. As the number of partitions increases, boundary nodes are replicated, often surpassing the number of inner nodes. This leads to significant communication overhead, with more time spent exchanging boundary node data, ultimately reducing training efficiency. As shown in the Table 1, the Vanilla method faces considerable communication costs. As dataset size and partitions grow, communication overhead becomes the primary bottleneck, significantly restricting training speed.

Long Communication Waiting Time Another significant challenge in partitioned parallel training is the long communication waiting time. As partitions increase, communication phases become more frequent and longer, leading to substantial idle times for GPUs. During these phases, the GPUs are often left waiting for boundary node information to be exchanged, resulting in poor utilization of computational resources. This not only reduces the overall training throughput but also exacerbates the latency between partitions, further slowing down the entire training process. As illustrated in the Fig. 1, whether it is Vanilla or PipeGCN, when communication time significantly exceeds computation time, there is a long interval between the computations of two consecutive iterations, resulting in substantial communication waiting time.

3.2 The Proposed PipeQS Method

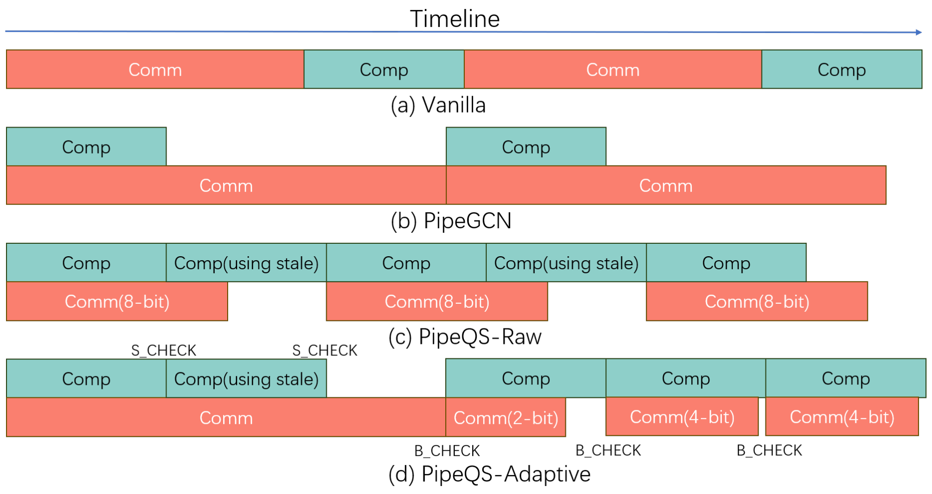
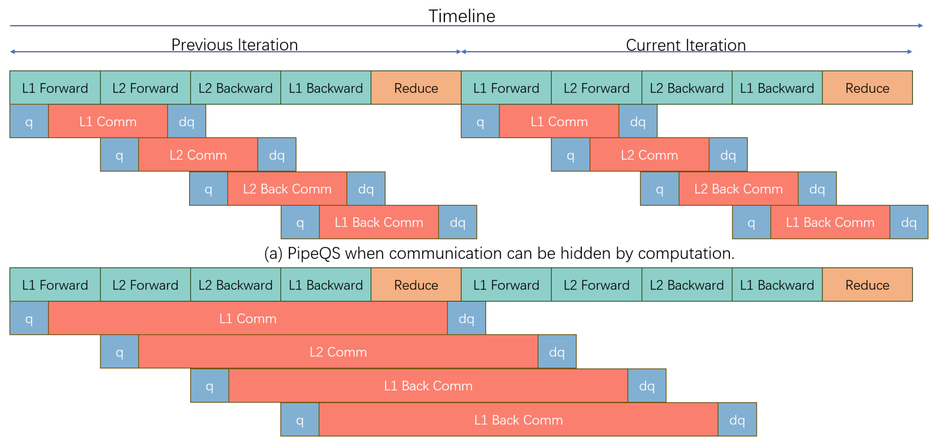


Fig. 1. Comparison of communication and computation for Vanilla, PipeGCN, PipeQS-Raw, and PipeQS-Adaptive on a single layer. The Vanilla method executes communication and computation sequentially. PipeGCN parallelizes them within a layer, but computation still waits for the previous iteration's communication. PipeQS-Raw reduces communication time through quantization and features/gradients staleness, skipping communication when it exceeds computation time. PipeQS-Adaptive dynamically adjusts bit-width and staleness for further optimization. In PipeQS-Adaptive, S_CHECK determines whether communication should be skipped and stale features/gradients should be used, while B_CHECK determines whether communication precision needs to be adjusted.

Fig. 1 presents a iteration-level overview of the PipeQS method, which pipelines the communication and computation phases. The figure also compares Vanilla and PipeQS with PipeGCN. Fig. 2 illustrates the finer-grained operation of PipeQS at the layer level.



(a) PipeQS when communication can be hidden by computation.

(b) PipeQS-Raw when communication cannot be hidden.

Fig. 2. Fine-grained layer-level operation of PipeQS. (a) shows the operation of PipeQS when communication is hidden by computation. (b) illustrates how PipeQS-Raw uses stale features/gradients for computation when communication cannot be hidden.

PipeGCN breaks the synchronization between communication and computation, allowing each GNN layer's computation to overlap with communication, thereby reducing the heavy overhead seen in vanilla methods. However, for large-scale graph training, communication time often far exceeds computation time, leaving a significant portion of communication time uncovered. Therefore, PipeGCN fails to address the two issues we mentioned earlier in distributed training.

PipeQS addresses the two problems through two key strategies respectively. First, it employs quantization to reduce communication time by decreasing the size of transmitted data. Second, it leverages stale features or gradients while parallelizing communication and computation, allowing training to continue without waiting for communication to finish by using features or gradients from the previous iteration. We refer to the method with fixed quantization bit-width and unlimited staleness as PipeQS-Raw. The implementation is shown in Alg. 1, and the complete process can be found in Alg. D.1 in Appendix D.

While effective, these techniques inevitably introduce certain challenges. Quantization reduces the precision of the transmitted features and gradients, and skipping communication introduces staleness, leading to a mix of up-to-date internal features and gradients with stale boundary features and gradients. To alleviate these effects and ensure convergence, PipeQS incorporates a bit-width and staleness adjustment strategy that dynamically adjusts the bit-width for quantization and determines when to apply staleness, allowing the training process to balance communication efficiency and accuracy.

Algorithm 1 PipeQS-Raw Forward Propagation (Per-Partition View)

```
Require: partition id $i$, partition count $n$, graph partition $G_i$, propagation matrix $P_i$,
    boundary node set $B_i$, layer count $L$, initial model $W_0$
Ensure: trained model $W_T$ after $T$ iterations
 1: $H_{prev} \leftarrow 0$                                  ▷ initialize previous communication result
 2: for $t := 1 \rightarrow T$ do
 3:     for $\ell := 1 \rightarrow L$ do
 4:         if $t > 1$ and $thread_f^{(\ell)}$ not completed then
 5:             Use $H_{prev}$                                ▷ skip communication
 6:         else
 7:             with $thread_f^{(\ell)}$                      ▷ communicate features in parallel
 8:                 Quantize $H_{S_{i,:}}^{(\ell)}$ to $H_{S_{i,:}}^{(\ell),q}$ with $Q_{S_{i,:}}^{(\ell)}$
 9:                 Send $H_{S_{i,:}}^{(\ell),q}$ and $Q_{S_{i,:}}^{(\ell)}$; Receive $B_{:}^{(\ell),q}$ and $Q_{B_:}^{(\ell)}$
10:                 Dequantize $B_{:}^{(\ell),q}$ with $Q_{B_:}^{(\ell)}$; Update $H_{prev}$
11:         end if
12:         $H_{V_i}^{(\ell)} \leftarrow \sigma(P_i H^{(\ell-1)} W^{(\ell-1)})$    ▷ update inner nodes
13:     end for
14:     Backward propagation and update model
15: end for
16: return $W_T$
```

3.3 Stochastic Integer Quantization

We adopt a stochastic integer quantization method to reduce communication overhead by converting high-precision floating-point data into lower bit-width integers (e.g., 2-bit, 4-bit, or 8-bit) via scaling and stochastic rounding [4]. In our adaptive approach, each tensor is quantized independently. For every tensor, we first compute its minimum (min) and maximum (max) values to derive the scaling factor:

$$\text{scale} = \frac{2^{\text{bits}} - 1}{\max - \min}, \tag{3}$$

which linearly maps the original floating-point data into the quantized integer domain. The tensor-specific parameters min and scale are stored individually and updated at every training batch to adapt to changes in the data distribution.

The quantization operation is defined as:

$$q(x) = \text{round}\left(\frac{x - \min}{\text{scale}} + \text{noise}\right), \tag{4}$$

where x represents the floating-point input data. The noise term facilitates stochastic rounding to reduce quantization bias. After quantization, the resulting integers are packed into compact bit-streams for efficient communication; the process is highly parallelized using CUDA to ensure scalability with large datasets.

Upon transmission, dequantization is performed using the stored min and scale values:

$$x_{\text{recovered}} = q(x) \times \text{scale} + \min . \tag{5}$$

Leveraging AdaQP-inspired parallel processing [19], our approach minimizes the overhead of quantization and dequantization while preserving model accuracy, thereby enhancing the scalability and performance of distributed GNN training.

3.4 Stale Features and gradients Utilization

In our distributed GNN communication protocol, we incorporate a staleness mechanism that allows the use of outdated neighbor node features and gradients to reduce communication overhead.

This technique will be used when the current round of computation finishes before the previous communication round completes. When staleness is permitted, rather than waiting for the most up-to-date neighbor features and gradients to arrive, the system proceeds with the outdated data from the previous communication. This enables uninterrupted computation, reducing the idle time spent waiting for communication to complete. By doing so, we effectively skip sending new communication messages for the current round, as the staleness mechanism ensures that previously received data can still be utilized in the ongoing computation.

This approach significantly improves overall efficiency in distributed environments, where communication latency often becomes a bottleneck.

3.5 Bit-width and Staleness Adjustment Strategy

To balance the trade-off between communication overhead and model accuracy, we propose a bit-width and staleness adjustment strategy. This strategy dynamically adjusts the bit-width for quantization and decides whether to use stale features/gradients at each layer during training.

For each GNN layer ℓ, we measure the difference between the current features/gradients n^ℓ and the previous features/gradients o^ℓ using inverted cosine similarity:

$$d^\ell = 1 - \frac{o^\ell \cdot n^\ell}{\|o^\ell\| \|n^\ell\|} \tag{6}$$

If the difference $d^\ell > E$, staleness is disabled, meaning that stale data cannot be reused ($r^\ell = 0$), and fresh communication must occur. Additionally, to ensure communication precision, we dynamically adjust the bit-width based on d^ℓ. This adjustment is governed by a logistic function:

$$p^\ell = \frac{1}{1 + \exp\left(-K \cdot \frac{d^\ell}{q^\ell}\right)} \tag{7}$$

Here, q^ℓ is a counter tracking the continuous quantization updates and the staleness count, and K is a scaling factor that controls the sensitivity of the adjustment. If $p^\ell > 0.5$, indicating a significant difference between iterations, the bit-width is increased to enhance precision. Conversely, if $p^\ell \leq 0.5$, the bit-width is reduced, thereby speeding up communication to avoid staleness or communication delays.

On the other hand, if the difference $d^\ell \leq E$, staleness is enabled ($r^\ell = 1$), and the previous features/gradients can be reused without adjusting the bit-width, effectively bypassing communication and improving efficiency.

As shown in Fig. 1, the aforementioned strategy is applied during the B_CHECK phase. Additionally, during each computation step, if communication has not yet completed, an S_CHECK is performed. The result of the S_CHECK determines whether to skip communication, based on the staleness flag r^ℓ obtained from the previous B_CHECK.

This adaptive strategy effectively reduces communication time while limiting the error within the threshold E, ensuring convergence and efficient training. The specific code for the strategy can be found in Alg. D.2 in Appendix D. We refer to the version of PipeQS that applies this strategy as PipeQS-Adaptive, with its implementation provided in Alg. D.3 in Appendix D.

3.6 Convergence Guarantee for PipeQS

Due to the quantization and staleness of neighboring node features and gradients, the convergence behavior of PipeQS-Adaptive requires theoretical justification. Unlike traditional full-precision synchronous updates, the presence of quantization noise and staleness error introduces additional challenges in the optimization process.

We establish the following convergence bound for PipeQS-Adaptive:

$$\frac{1}{T}\sum_{t=1}^{T}\mathbb{E}[\|\nabla L(W_t)\|^2] \leq O(T^{-1/2}) + O(\sigma_q^2 + E^2) \tag{8}$$

where $L(W)$ represents the loss function, and σ_q, E denote the quantization noise and staleness error, respectively. The final convergence rate is $O(T^{-1/2})$, but it may be affected by quantization and staleness effects. Hence, an adaptive strategy for adjusting bit-width and staleness threshold is crucial to ensuring efficient convergence while maintaining a reasonable error bound.

The detailed proof of this convergence result is provided in Appendix C.

4 Experiment

4.1 Implementation

We implement PipeQS on top of DGL 2.3.0 [22] and PyTorch 2.3.0 [14]. DGL is utilized for graph-related data storage and operations, while PyTorch's distributed package is employed for initializing process groups and facilitating communication between devices. Before training begins, the graph is partitioned using DGL's built-in METIS algorithm [9]. Each training process is confined to a single device (GPU), and remote node indices—derived from DGL's partition book—are broadcast to create sending and receiving node index sets, allowing processes to fetch the required messages from other devices efficiently.

4.2 Experimental Settings

We evaluated PipeQS on three large benchmark datasets, namely Reddit, Ogbn-products, and Yelp [5, 7, 26]. The transductive graph learning tasks on Reddit and Ogbn-products involve single-label node classification, while multi-label classification tasks are performed on Yelp. We use accuracy and F1-score (micro) as the performance metrics for these tasks, collectively referred to as accuracy. All datasets follow the "fixed-partition" splits using the METIS algorithm. We train the GCN model [10] for all experiments, ensuring consistency by unifying all model-related and training-related hyperparameters. The model layer size is set to 3, with hidden layers of 256 dimensions, and the learning rate is fixed at 0.01. We employ the Adam optimizer and use the ReLU activation function within the GCN model. For PipeQS using adaptive strategy, we fix the threshold E to 0.1.

Experiments are conducted on a server running Ubuntu 20.04 LTS, equipped with an Intel(R) Xeon(R) Platinum 8362 CPU, 360GB of RAM, 8 NVIDIA RTX 3090 (24GB) GPUs and PCIe4.0x16 connecting CPU-GPU and GPU-GPU.

4.3 Comparative Performance Evaluation

In this section, we evaluate the performance of PipeQS-Adaptive in comparison with Vanilla GCN and three other state-of-the-art (SOTA) methods: PipeGCN [21], SANCUS [15], and AdaQP [19]. The datasets used for this comparison include Reddit, Yelp, and Ogbn-products, which provide diverse scenarios for testing both communication efficiency and model convergence.

PipeGCN optimizes training by parallelizing communication and computation, relying on stale features/gradients from the previous iteration to reduce idle communication time. AdaQP focuses on minimizing communication overhead by quantizing the transmitted features/gradients, thereby lowering data transfer costs. SANCUS adopts a staleness-aware strategy, caching and reusing stale embeddings to skip unnecessary broadcasts, which reduces communication while maintaining convergence through bounded approximation errors. Our approach, PipeQS-Adaptive, leverages adaptive methods to dynamically adjust parameters for optimal performance under varying conditions. For a detailed methodological comparison between PipeQS-Adaptive and PipeQS-Raw, as well as an analysis of the impact of bit-width and staleness, please refer to Appendix B.

Table 2. Training time, throughput, and communication overhead comparison on different datasets

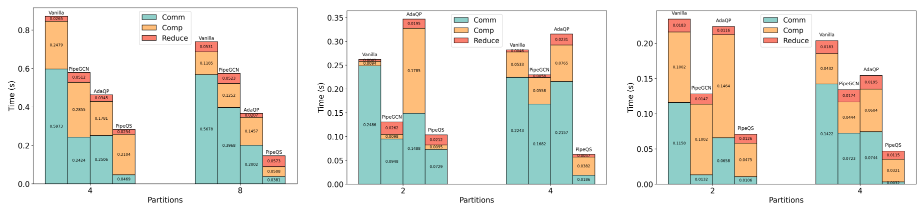
Dataset	Partitions	Method	Comm.(s)	Time(s)	Comm./Time (%)	Throughput (epoch/s)	Speedup (x)
Reddit	2	Vanilla	0.2486	0.2621	94.85%	3.8153	1.00x
		PipeGCN	0.0948	0.1308	72.47%	7.6453	2.00x
		SANCUS	**0.0286**	0.1887	**15.15%**	5.2994	1.39x
		AdaQP	0.1488	0.3468	42.90%	2.8835	0.76x
		PipeQS	0.0729	**0.1036**	70.37%	**9.6525**	**2.53x**
	4	Vanilla	0.2243	0.2822	79.47%	3.5436	1.00x
		PipeGCN	0.1682	0.2298	73.20%	4.3516	1.23x
		SANCUS	0.3879	0.5246	73.94%	1.9062	0.54x
		AdaQP	0.2157	0.3153	68.42%	3.1716	0.89x
		PipeQS	**0.0186**	**0.0625**	**29.76%**	**16.0000**	**4.51x**
Yelp	2	Vanilla	0.1158	0.2343	49.42%	4.2680	1.00x
		PipeGCN	0.0132	0.1281	**10.30%**	7.8064	1.83x
		AdaQP	0.0658	0.2238	29.40%	4.4683	1.05x
		PipeQS	**0.0106**	**0.0707**	14.99%	**14.1443**	**3.31x**
	4	Vanilla	0.1422	0.2037	69.82%	4.9092	1.00x
		PipeGCN	0.0723	0.1341	53.94%	7.4571	1.52x
		AdaQP	0.0744	0.1543	48.22%	6.4809	1.32x
		PipeQS	**0.0032**	**0.0468**	**6.84%**	**21.3675**	**4.35x**
Ogbn-products	4	Vanilla	0.5973	0.8717	68.53%	1.1472	1.00x
		PipeGCN	0.2424	0.5791	41.85%	1.7268	1.51x
		SANCUS	1.2423	1.6863	73.67%	0.5930	0.52x
		AdaQP	0.2506	0.4632	54.12%	2.1589	1.88x
		PipeQS	**0.0469**	**0.2827**	**16.59%**	**3.5373**	**3.08x**
	8	Vanilla	0.5678	0.7394	76.79%	1.3524	1.00x
		PipeGCN	0.3968	0.5743	69.07%	1.7413	1.29x
		SANCUS	1.5913	1.8536	85.88%	0.5394	0.40x
		AdaQP	0.2002	0.3666	54.61%	2.7278	2.02x
		PipeQS	**0.0381**	**0.1462**	**26.06%**	**6.8399**	**5.06x**

Training Time and Throughput Comparison Table 2 compares the training time and throughput of PipeQS against Vanilla GCN, PipeGCN, SANCUS, and AdaQP across three datasets.

As shown in Table 2, on Reddit, PipeQS is 2.53× to 4.51× faster than Vanilla and up to 1.45× faster than PipeGCN. For Yelp, PipeQS achieves a 3.31× to 4.35× speedup over Vanilla and is up to 2.86× faster than PipeGCN. On Ogbn-products, PipeQS shows the most significant improvement, with a 3.08× to 5.06× speedup over Vanilla and up to 3.93× faster than PipeGCN. Overall, PipeQS delivers the best throughput across all configurations.

Communication Overhead Comparison PipeQS's reduction in training time primarily stems from the decrease in both communication overhead and communication waiting time. Table 2 and Fig. 3 analyze the communication overhead across different methods and datasets. PipeQS consistently reduces communication overhead, especially with increased partitions. For instance, on the Ogbn-products dataset with 8 partitions, PipeQS lowers communication overhead to 26.06% of total training time, compared to 76.79% for Vanilla GCN.

While methods such as PipeGCN, AdaQP, and SANCUS introduce different techniques to address communication issues, they fall short in scalability on larger datasets. PipeGCN hides communication within computation but cannot sustain this as datasets or partition numbers grow, leading to a steep rise in communication overhead. AdaQP reduces communication overhead through quantization but fails to address communication waiting, resulting in a signif-



(a) Ogbn-products **(b)** Reddit **(c)** Yelp

Fig. 3. Training time breakdown of Vanilla GCN, PipeGCN, AdaQP, and PipeQS.

icant increase in communication time for larger datasets. Similarly, SANCUS leverages staleness to reduce communication waiting time, but its performance deteriorates in larger-scale datasets, as evidenced by the increasing communication time with more partitions.

In contrast, PipeQS not only reduces communication overhead through quantization but also addresses communication waiting by leveraging a combination of pipelining and staleness. This approach ensures consistently low communication time across all datasets and partition sizes, showcasing its superior scalability and efficiency, particularly in large-scale distributed GNN training scenarios where other methods struggle.

Accuracy Comparison PipeQS reduces training time while still maintaining high accuracy. Table 3 summarizes the final test accuracy of different methods across the datasets. As shown, PipeQS consistently achieves accuracy comparable to that of Vanilla GCN and, in some cases, even surpasses other methods. For example, in the 4-partition setup on the Yelp dataset, PipeQS reaches an accuracy of 47.34%, significantly outperforming Vanilla GCN's 45.19%, which demonstrates the effectiveness of the quantization and staleness adjustment strategies in preserving model accuracy.

Other methods, such as AdaQP, perform well on smaller datasets like Reddit, where communication overhead is lower and the variance introduced by quantization is minimal. However, as the dataset size increases, the variance error caused by quantization begins to hinder convergence, leading to a drop in accuracy. This is particularly evident in larger datasets like Ogbn-products, where AdaQP's performance starts to deteriorate.

In contrast, although PipeQS's accuracy is slightly lower than AdaQP in certain settings on the Ogbn-products dataset, its overall performance remains near-optimal. This confirms that PipeQS effectively balances accuracy and training efficiency, showing resilience in larger datasets.

Convergence Speed Comparison To evaluate convergence speed, we analyze the relationship between accuracy and the number of epochs for each method.

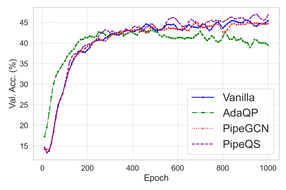
Table 3. Accuracy comparison on different datasets

Dataset	Partitions	Vanilla	PipeGCN	SANCUS	AdaQP	PipeQS
Reddit	2	94.85	94.63	94.15	**95.41**	94.78
	4	94.96	94.68	94.17	**95.34**	94.51
Yelp	2	46.15	44.88	-	43.43	**47.11**
	4	45.19	44.63	-	43.38	**47.34**
Ogbn-products	4	76.04	76.36	71.52	75.59	**77.04**
	8	75.47	76.71	71.99	75.24	**76.75**

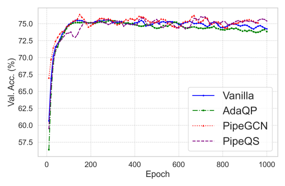
The results indicate that PipeQS generally exhibits better convergence across multiple datasets. As shown in Fig. 4, in the 8-partition setup on the Ogbn-products dataset, PipeQS surpasses 76% validation accuracy within 200 epochs. In comparison, Vanilla GCN and other methods either require more epochs to reach a similar accuracy level or fail to reach it entirely.

In the time-accuracy relationship analysis (Fig. 5), PipeQS demonstrates outstanding training efficiency. Across all datasets and partition settings, PipeQS's accuracy increases significantly faster than that of other methods. For instance, in the 2-partition experiment on the Yelp dataset, PipeQS achieves close to 45% validation accuracy in just under 50 seconds, whereas other methods requires much more time to approach this level.

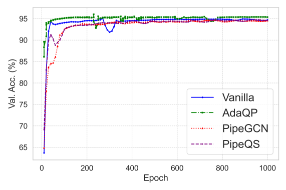
These results highlight that PipeQS not only improves training efficiency but also effectively maintains model accuracy, while offering significant advantages in terms of training time.



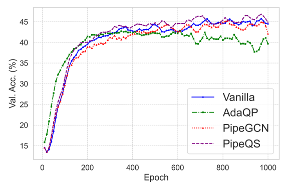
(a) Yelp - 2



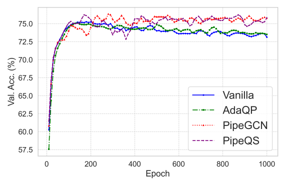
(c) Ogbn-products - 4



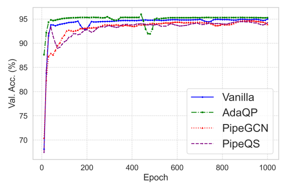
(e) Reddit - 2



(b) Yelp - 4

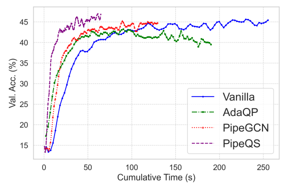


(d) Ogbn-products - 8

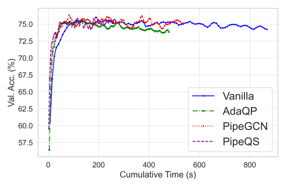


(f) Reddit - 4

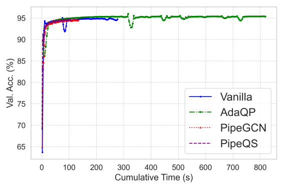
Fig. 4. Epoch-Accuracy curves for different datasets and partition settings.



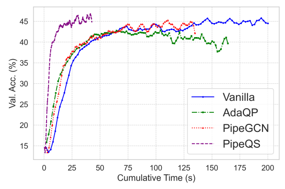
(a) Yelp - 2



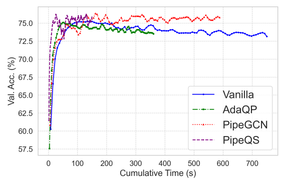
(c) Ogbn-products - 4



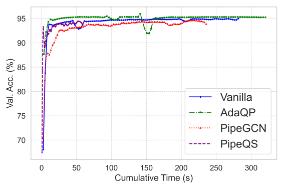
(e) Reddit - 2



(b) Yelp - 4



(d) Ogbn-products - 8



(f) Reddit - 4

Fig. 5. Time-Accuracy curves for different datasets and partition settings.

5 Conclusion

Training GNNs on large-scale graphs in distributed environments presents a significant challenge, primarily due to the substantial communication overhead and prolonged communication waiting times. Existing distributed GNN training methods, such as sampling and pipelining techniques, partially alleviate these issues but still suffer from high communication costs, synchronization delays, and unstable convergence, especially as the graph size increases and inter-node boundary communication becomes more costly. While recent methods incorporating quantization and staleness have reduced communication volume, they typically sacrifice convergence stability and accuracy, and there remains a lack of effective solutions to fully eliminate communication waiting time.

In this work, we propose PipeQS, a novel distributed GNN training method that integrates quantization and staleness within a pipeline framework. We introduce an adaptive mechanism for adjusting bit-width and staleness iterations, ensuring communication efficiency without compromising convergence stability, maintaining a convergence rate of $O(T^{-\frac{1}{2}})$. Our theoretical analysis and empirical evaluations demonstrate that PipeQS achieves significant improvements in training efficiency across various large-scale graph datasets, while preserving stable convergence. Looking ahead, we plan to extend PipeQS to heterogeneous and temporal graphs, further exploring its adaptability and performance in more complex graph structures.

Acknowledgements

The authors would like to thank the anonymous reviewers for their helpful comments. This work was supported by the National Natural Science Foundation of China (Grant Nos. 62372391, 62072384); the Fujian Provincial Major Science and Technology Project (2022YZ040011); the Wang Deyao Outstanding Graduate Scholarship Program of Xiamen University; and the National Key Research and Development Program of China (Grant Nos. 2023YFF1205600, 2024YFF1206204).

References

1. Chen, J., Zhu, J., Song, L.: Stochastic training of graph convolutional networks with variance reduction. In: Proceedings of the 35th International Conference on Machine Learning (ICML). pp. 942–950 (2018)
2. Chiang, W.L., Liu, X., Si, S., Li, Y., Bengio, S., Hsieh, C.J.: Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD). pp. 257–266 (2019)
3. Feng, B., Wang, Y., Chen, G., et al.: Sgquant: Squeezing the last bit on graph neural networks with specialized quantization. In: Proceedings of the 32nd International Conference on Tools with Artificial Intelligence (ICTAI) (2020)
4. Gupta, S., Agrawal, A., Gopalakrishnan, K., Narayanan, P.: Deep learning with limited numerical precision. In: Proceedings of the 32nd International Conference on Machine Learning (ICML). pp. 1737–1746 (2015)
5. Hamilton, W., Ying, Z., Leskovec, J.: Inductive representation learning on large graphs. In: Advances in Neural Information Processing Systems (NIPS). vol. 30 (2017)
6. Ho, Q., Cipar, J., Cui, H., Lee, S., Kim, J.K., Gibbons, P.B., Gibson, G.A., Ganger, G.R., Xing, E.P.: More effective distributed ml via a stale synchronous parallel parameter server. In: Advances in Neural Information Processing Systems (NIPS). vol. 26 (2013)
7. Hu, W., Fey, M., Zitnik, M., Dong, Y., Ren, H., Liu, B., Catasta, M., Leskovec, J.: Open graph benchmark: Datasets for machine learning on graphs. Advances in Neural Information Processing Systems **33**, 22118–22133 (2020)
8. Jia, Z., Lin, S., Gao, M., Zaharia, M., Aiken, A.: Improving the accuracy, scalability, and performance of graph neural networks with roc. Proceedings of Machine Learning and Systems **2**, 187–198 (2020)
9. Karypis, G., Kumar, V.: A fast and high-quality multilevel scheme for partitioning irregular graphs. SIAM Journal on Scientific Computing **20**(1), 359–392 (1998)
10. Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. In: Proceedings of the International Conference on Learning Representations (ICLR) (2017)
11. Li, M., Andersen, D.G., Park, J.W., Smola, A.J., Ahmed, A., Josifovski, V., Long, J., Shekita, E.J., Su, B.Y.: Scaling distributed machine learning with the parameter server. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI). pp. 583–598 (2014)
12. Ma, L., Yang, Z., Miao, Y., Xue, J., Wu, M., Zhou, L., Dai, Y.: Neugraph: Parallel deep neural network computation on large graphs. In: 2019 USENIX Annual Technical Conference (USENIX ATC). pp. 443–458 (2019)
13. Niu, F., Recht, B., Ré, C., Wright, S.J.: Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In: Advances in Neural Information Processing Systems (NIPS). vol. 24 (2011)

14. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al.: Pytorch: An imperative style, high-performance deep learning library. In: Advances in Neural Information Processing Systems (NeurIPS). vol. 32 (2019)
15. Peng, J., Chen, Z., Shao, Y., Shen, Y., Chen, L., Cao, J.: Sancus: Staleness-aware communication-avoiding full-graph decentralized training in large-scale graph neural networks. Proceedings of the VLDB Endowment **15**(9), 1937–1950 (2022)
16. Song, Z., Gu, Y., Qi, J., Wang, Z., Yu, G.: Ec-graph: A distributed graph neural network system with error-compensated compression. In: Proceedings of the 38th IEEE International Conference on Data Engineering (ICDE). pp. 648–660 (2022)
17. Thorpe, J., Qiao, Y., Eyolfson, J., Teng, S., Hu, G., Jia, Z., Wei, J., Vora, K., Netravali, R., Kim, M., et al.: Dorylus: Affordable, scalable, and accurate gnn training with distributed cpu servers and serverless threads. In: 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI). pp. 495–514 (2021)
18. Tripathy, A., Yelick, K., Buluç, A.: Reducing communication in graph neural network training. In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–14. IEEE (2020)
19. Wan, B., Zhao, J., Wu, C.: Adaptive message quantization and parallelization for distributed full-graph gnn training. Proceedings of Machine Learning and Systems **5** (2023)
20. Wan, C., Li, Y., Li, A., Kim, N.S., Lin, Y.: Bns-gcn: Efficient full-graph training of graph convolutional networks with partition-parallelism and random boundary node sampling. Proceedings of Machine Learning and Systems **4**, 673–693 (2022)
21. Wan, C., Li, Y., Wolfe, C.R., Kyrillidis, A., Kim, N.S., Lin, Y.: Pipegcn: Efficient full-graph training of graph convolutional networks with pipelined feature communication. In: Proceedings of the 39th International Conference on Machine Learning (ICML) (2022)
22. Wang, M., Yu, L.: Deep graph library: Towards efficient and scalable deep learning on graphs. In: ICLR Workshop on Representation Learning on Graphs and Manifolds (2019)
23. Xu, K., Hu, W., Leskovec, J., Jegelka, S.: How powerful are graph neural networks? In: Proceedings of the International Conference on Learning Representations (ICLR) (2019)
24. Yang, H.: Aligraph: A comprehensive graph neural network platform. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD). pp. 3165–3166 (2019)
25. Ying, R., He, R., Chen, K., Eksombatchai, P., Hamilton, W.L., Leskovec, J.: Graph convolutional neural networks for web-scale recommender systems. In: Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD). pp. 974–983 (2018)
26. Zeng, H., Zhou, H., Srivastava, A., Kannan, R., Prasanna, V.K.: Graphsaint: Graph sampling based inductive learning method. In: Proceedings of the International Conference on Learning Representations (ICLR) (2020)
27. Zhang, M., Chen, Y.: Link prediction based on graph neural networks. In: Advances in Neural Information Processing Systems (NeurIPS). vol. 31 (2018)