BOOSTING DISTRIBUTED FULL-GRAPH GNN TRAINING WITH ASYNCHRONOUS ONE-BIT COMMUNICATION

Meng Zhang ¹ Qinghao Hu ¹² Peng Sun ²³ Yonggang Wen ¹ Tianwei Zhang ¹

ABSTRACT

Training Graph Neural Networks (GNNs) on large graphs is challenging due to the conflict between the high memory demand and limited GPU memory. Recently, distributed full-graph GNN training has been widely adopted to tackle this problem. However, the substantial inter-GPU communication overhead can cause severe throughput degradation. Existing communication compression techniques mainly focus on traditional DNN training, whose bottleneck lies in synchronizing gradients and parameters. We find they do not work well in distributed GNN training as the barrier is the layer-wise communication of features during the forward pass & feature gradients during the backward pass. To this end, we propose an efficient distributed GNN training framework **Sylvie**, which employs one-bit quantization technique in GNNs and further pipelines the curtailed communication with computation to enormously shrink the overhead while maintaining the model quality. In detail, Sylvie provides a lightweight *Low-bit Module* to quantize the sent data and dequantize the received data back to full precision values in each layer. Additionally, we propose a *Bounded Staleness Adaptor* to control the introduced staleness to achieve further performance enhancement. We conduct theoretical convergence analysis and extensive experiments on various models & datasets to demonstrate Sylvie can considerably boost the training throughput by up to 28.1×.

1 Introduction

In recent years, GNNs have become very popular and showed state-of-the-art (SOTA) performance in learning structured data like graphs. GNNs capture the underlying dependencies of the given graph via message passing operations (Abadal et al., 2021). To update nodes, GNNs first aggregate the feature vectors from the nodes' neighbors and then combine them together. Despite their impressive performance on graph-related tasks, training GNNs on large-scale graphs containing millions to billions of nodes is still a long-standing issue, as extensive memory resources are needed for loading and computing input graphs (Kipf & Welling, 2016; Liu et al., 2021; Zhang & Chen, 2018), hindering the practical development of more sophisticated GNN models.

Existing solutions to this problem can be divided into two directions. First, some works (Chen et al., 2018; Chiang et al., 2019; Huang et al., 2018; Zeng et al., 2020; Zou et al., 2019) propose sampling-based methods which only select a subset of nodes and edges to be trained at each iteration. However, most of these methods need centralized data storage, which will cause significant data transfer costs. More importantly,

Preprint.

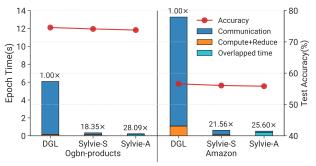


Figure 1. Epoch time and test accuracy of training GCN on two datasets with DGL, Sylvie-S and Sylvie-A over two servers. Sylvie provides up to $28.1 \times$ speedup.

these methods suffer from model accuracy loss (Hamilton et al., 2017; Jia et al., 2020; Wan et al., 2022a). Table 1 shows the test accuracy of the GraphSAGE model on the Ogbn-products dataset (Hu et al., 2020) when training with the sampling-based mode and full-graph. The accuracy of sampling-based mode is always lower than that of full-graph training, especially as the sample size decreases.

Distributed GNN training (Thorpe et al., 2021; Zheng et al., 2021; Zhu et al., 2019) is another promising direction, which conducts full graph training on multiple GPUs or nodes to reduce the computing time and memory demand on each GPU. It first splits the whole graph into several subgraphs so that each can fit in a single GPU, and then trains sub-

¹Nanyang Technological University ²Shanghai AI Laboratory ³SenseTime Research. Correspondence to: Tianwei Zhang <tianwei.zhang@ntu.edu.sg>.

Table 1. Test accuracy of full-graph training and sampling-based training with different numbers of neighborhood samples in each layer when training GraphSAGE on Ogbn-products.

	San	pling-b	Full-Graph	
Sample Size	5	10	15	run orupn
Accuracy (%)	73.55	74.87	76.84	79.19

graphs on each GPU locally. Current GNN frameworks like DGL (Wang et al., 2020) and PyG (Fey & Lenssen, 2019) have the scalability limitation and lack system support for distributed full-graph training. In addition, unlike classical distributed DNN training (e.g., image classification with AlexNet (Wen et al., 2017)) where training samples are independent of each other, it is non-trivial to apply data parallelism on GNNs due to the node dependency between subgraphs, leading to obligatory data communication overhead. To solve this, recent works (Jia et al., 2020; Mostafa, 2022; Tripathy et al., 2020; Wan et al., 2022a;b) proposed various approaches to realize efficient distributed full-graph training. However, these approaches also exhibit several drawbacks. Prior methods (Jia et al., 2020; Mostafa, 2022; Tripathy et al., 2020) still suffer from heavy communication overhead, resulting in considerable training time. Some works (Mostafa, 2022; Wan et al., 2022a) train large-scale graphs at the price of potential accuracy loss. Moreover, it is hard to generalize some approaches (Tripathy et al., 2020; Wan et al., 2022a;b) to various GNN models or datasets.

To cope with the aforementioned limitations, we propose a new framework Sylvie to effectively accelerate distributed full-graph GNN training while maintaining the model accuracy. Sylvie consists of five main modules: (1) Graph Engine first partitions the original graph to GPUs and selects the communicated nodes. (2) Quantizer compresses the communicated features and gradients. (3) Communicator transfers data between partitions. (4) Dequantizer decompresses the received data in an error-compensated way. (5) Trainer finally conducts model forward and backward computation using the communicated data. We also introduce two variants of Sylvie: (1) Sylvie-S executes the above modules sequentially. (2) Sylvie-A integrates the asynchronous pipeline technique to further improve the performance. We also design a *Bounded Staleness Adaptor* for Sylvie-A to enhance the model convergence in some cases where convergence is affected by stale features & gradients. Fig.1 shows part of the results when training with Sylvie on two machines. Sylvie dramatically reduces the training time with minor accuracy loss. The extensive communication overhead (blue bar) is cut down to a very tiny portion, thus contributing to the significant speedup on the epoch time. On top of Sylvie-S, the curtailed communication is further overlapped by Sylvie-A, achieving up to 28.1× speedup. In addition to the scalability over the multi-server setting, Sylvie also achieves great performance on a single machine: e.g., up to $9.3 \times$ improvement over DGL.

In summary, we make the following contributions:

- To the best of our knowledge, Sylvie is the *first* framework to implement one-bitwidth communication quantization to accelerate multi-GPU GNN training, achieving up to 28.1× speedup over existing GNN frameworks.
- Based on our plentiful experiments and theoretical analysis, Sylvie can well maintain the model convergence with negligible accuracy loss.
- Beyond the *spacial* size reduction, we also consider the orthogonal *temporal* aspect by integrating the asynchronous pipeline technique (Sylvie-A). Additionally, for better model convergence, we design a *Bounded Staleness Adaptor* to reduce the errors incurred by stale data.

2 BACKGROUND AND RELATED WORK

2.1 Graph Neural Networks

GNNs are machine learning algorithms that learn from the graph connectivity and model the relationship between nodes. In general, the iterative learning process contains two steps in each layer: feature aggregation and update. Consider a graph $\mathcal{G}=(V,E)$ with nodes $V=\{v_1,\cdots,v_{|V|}\}$, edges $E=\{e_1,\cdots,e_{|E|}\}$ and node feature matrix $\boldsymbol{X}\in\mathbb{R}^{|V|\times d}$. For an arbitrary layer $l\in[1,L]$, the aggregation and update steps can be expressed as:

$$\boldsymbol{z}_{v}^{(l)} = \rho^{(l)} \left(\left\{ \boldsymbol{h}_{u}^{(l-1)} \mid u \in \mathcal{N}(v) \right\} \right)$$
 (1)

$$\boldsymbol{h}_{v}^{(l)} = \phi^{(l)} \left(\boldsymbol{z}_{v}^{(l)}, \boldsymbol{h}_{v}^{(l-1)} \right)$$
 (2)

where $\mathcal{N}(v)$ is the neighboring nodes of node v. The aggregation function $\rho^{(l)}$ takes the embeddings of neighboring nodes $\boldsymbol{h}_u^{(l-1)}$ to get an intermediate aggregated result $\boldsymbol{z}_v^{(l)}$, which then serves as the input to update function $\phi^{(l)}$ together with the feature vector $\boldsymbol{h}_v^{(l-1)}$ of node v itself to obtain the learned embedding $\boldsymbol{h}_v^{(l)}$ at the l-th layer. Different GNNs vary in their aggregation and update functions. In this work, we mainly focus on three popular GNN models: GraphSAGE (Hamilton et al., 2017), GCN (Kipf & Welling, 2016) and GAT (Veličković et al., 2018), but our framework can be easily extended to other GNN models.

2.2 Distributed GNN Training

To conduct distributed GNN training on full graphs, the whole input graph is first partitioned on the host side to fit into a single GPU. Since each node and its features will only be assigned to one GPU, there are some nodes, dubbed **HALO nodes**, that are connected to nodes in the local partition but do not belong to this partition. As depicted by Fig.3, for the partition on GPU-1, node 4 requires extra features

of node 7 from GPU-0 and node 1 from GPU-2 to update its embedding in every layer. In the backward pass, the feature gradients computed locally are also incomplete. Since nodes only exchange their 1-hop neighbors in each layer, the dataflow from 2-hop to *L*-hop resides on other partitions. Therefore, the feature gradients of HALO nodes will also be broadcast in each layer. This communication overhead is non-trivial since the amount of HALO nodes can be excessive. In addition, such cost becomes more intensive as the number of partitions and layer size grow larger. For node 4 on GPU1, communication of nodes 7 and 1 should be done before layer1 begins. After layer1, the same process repeats for the rest layers in a sequential order strictly, as shown in Fig.4(a). In this case, excessive communication will take up the training time and block the subsequent computation.

To show the vast communication cost more intuitively, we profile the epoch time along with its breakdown on multiple datasets and models, as shown in Fig.2. We can clearly see for both GraphSAGE and GCN on three datasets, the communication time nearly dominates the entire training process (up to 89.23%), while the computation and gradient synchronization operation (all-reduce) only occupy a very small portion. The scalability and efficiency of distributed GNN training is thus seriously restrained due to the excessive communication overhead.

Prior works propose new frameworks to accelerate distributed GNN training, e.g., AliGraph (Zhu et al., 2019) and NeuGraph (Ma et al., 2019). However, these methods all store the partitions in CPUs, which inevitably incur frequent CPU-GPU swapping and largely impair the benefits of distributed training. DistDGL (Zheng et al., 2021) provides the scaling results but only on sampling-based methods. LLCG (Ramezani et al., 2022) totally drops dependent information between partitions and adds a global correction server to compensate for the error with redundant overhead. Moreover, those works only support mini-batch training on graphs rather than full-graph training.

Different from the above sampling-based works, ROC (Jia et al., 2020) accelerates distributed full-graph training, but it also stores the partitions in CPUs and suffers from the expensive CPU-GPU data transfer cost. SAR (Mostafa, 2022) provides memory savings for full-graph training in large scale, but it has more computation burden due to rematerialization. BNS-GCN (Wan et al., 2022a) adopts random sampling on the boundary nodes and shows impressive acceleration, yet it risks downgrading the model performance by dropping node connections and its performance is highly dependent on the graph structure. CAGNET (Tripathy et al., 2020) raises different dimension partitioning methods to boost training by slicing the node features to sub-vectors, at the extra communication and synchronization costs. PipeGCN (Wan et al., 2022b) ingeniously hides the communication

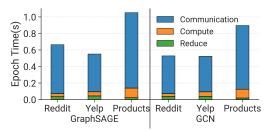


Figure 2. Training time per epoch and its breakdown in vanilla distributed GNN training with DGL on a single server (8 GPUs).

cost by pipelining computation and communication while keeping all the boundary information. However, its efficacy corrupts badly when the communication time is far larger than the computation part, as illustrated in §4.1.

2.3 Quantization for Neural Networks

Model quantization is commonly adopted to accelerate conventional DNN inference (Jacob et al., 2018; Krishnamoorthi, 2018), while we target training speedup. Compression has also been popular to reduce the communication and boost distributed DNN training (Alistarh et al., 2017; Wen et al., 2017). However, the overhead in distributed DNN training stems from synchronizing all gradients and parameters. Hence, they cannot be simply grafted to our scenario since the overhead of gradient synchronization like all-reduce is negligible in GNN training, as shown in Fig.2.

In recent years, some works (Feng et al., 2020; Zhao et al., 2020) apply GNN model quantization via simulation for memory reduction, with the underlying computation still in 32-bit full precision. A recent work EXACT (Liu et al., 2021) aims to reduce the memory demand at the cost of extra training time overhead, seriously deteriorating the training efficiency. Other works (Tailor et al., 2021; Wang et al., 2022) quantize GNN models for efficient inference. Compared with our work, all these methods have different targets and only consider small-scale datasets.

Different from the aforementioned quantization works, we are the **first** to explore the opportunity of quantizing communication in GNNs, which can substantially reduce the overhead with negligible accuracy loss. We do not quantize activations or weights like previous works (Jacob et al., 2018; Krishnamoorthi, 2018) because (1) computation only occupies a very small portion while communication of embeddings & feature gradients dominates the training overhead (Fig.2); (2) unique sparse computations in GNNs, e.g., SpMM in cuSPARSE (cuS, 2022), lack support for low precision computation, unlike dense operations (e.g. GEMM) in DNNs which support fast low precision computation.

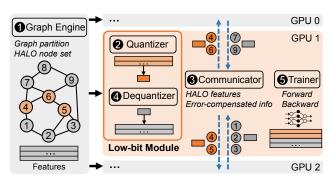


Figure 3. Overview of Sylvie framework & workflow. We focus on the subgraph on GPU-1. Orange nodes and rectangles represent nodes allocated to GPU-1 and their corresponding features. The others in gray represent nodes/features on other GPUs.

3 PROPOSED FRAMEWORK

3.1 Overview

Fig.3 depicts the overall workflow of Sylvie, consisting of five steps. **1** *Graph Engine* first partitions the input graph to several subgraphs on the host side, and constructs the HALO node set of each partition for the preparation of later communication. Then each partition, including the graph structure and features, is allocated to an individual GPU. 2 Next, Quantizer in each GPU quantizes the HALO node data into 1-bit integers. For instance, data of nodes 4, 5 & 6 in GPU-1 are quantized because they are required by other partitions in GPU-0 and GPU-1. 3 After this, those quantized data, along with the corresponding error-compensated information (used to help recover data), are broadcast between partitions through network communications. 4 After receiving the quantized data from other partitions, Dquantizer in each GPU recovers the compressed data back to full-precision values with the error-compensated information. Quantizer and Dequantizer together form the Low-bit Module that takes charge of data transformation. 6 Those recovered data, together with the original ones are utilized to train the model in both forward and backward passes.

We introduce two variants based on Sylvie (Fig.4). (1) **Sylvie-S**: all these modules are executed synchronously in a sequential order. (2) **Sylvie-A**: we adopt the asynchronous pipeline technique between *Low-bit Module*, communication and computation to further improve the training performance. However, this asynchronous pipeline can cause damages to the convergence on some datasets or models, due to the stale features and gradients. We further design a *Bounded Staleness Adaptor* to periodically synchronize the latest data for all partitions in a fixed number of epochs.

3.2 Sylvie Framework

Graph Engine. To conduct distributed GNN training, *Graph Engine* first partitions both the graph structure and feature matrix on the host side based on the number of avail-

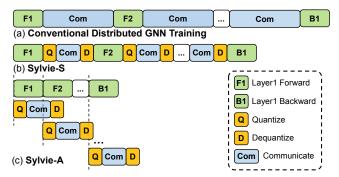


Figure 4. Workflow comparison between conventional distributed training, Sylvie-S and Sylvie-A. Sylvie reduces the massive communication overhead with *Low-bit Module*. Sylvie-A further curtails the training time by pipelining.

able GPUs. The HALO node set of each partition is also constructed. Then each partition is loaded to an individual GPU, and each GPU holds a replica of the full model.

Quantizer. Different from existing works that quantize all the activations or weights, Sylvie is the first to quantize the communication data in distributed GNN training. We illustrate the quantization mechanism using the example of the subgraph on GPU-1 during the forward pass in Fig.3. For the l-th layer of the model, Quantizer first selects node features needed for communication (e.g., features of nodes 4, 5 and 6) $S^{(l-1)} \subseteq H^{(l-1)}$ according to the HALO node set and then quantizes them to 1-bit integers $\hat{S}_{1bit}^{(l-1)}$ with low precision. For each node feature $h^{(l-1)}$ in $S^{(l-1)}$, to quantize it to a 1-bit integer, we use the following formula:

$$\bar{\boldsymbol{h}}^{(l-1)} = \frac{\boldsymbol{h}^{(l-1)} - \min(\boldsymbol{h}^{(l-1)})}{\max(\boldsymbol{h}^{(l-1)}) - \min(\boldsymbol{h}^{(l-1)})} B \qquad (3)$$

$$\hat{\boldsymbol{h}}_{1bit}^{(l-1)} = \left[\bar{\boldsymbol{h}}^{(l-1)}\right] \quad \text{w.prob.} \quad \bar{\boldsymbol{h}}^{(l-1)} - \left[\bar{\boldsymbol{h}}^{(l-1)}\right]$$
otherwise $\left[\bar{\boldsymbol{h}}^{(l-1)}\right] \qquad (4)$

where $B = 2^b - 1$ is the number of quantization bins when we quantize data to *b*-bit integers, and the default value of *b* is 1 in our work since we make 1-bit quantization. Equ.4 is the stochastic rounding (Courbariaux et al., 2015).

Communicator. After the quantization, $\hat{S}^{(l-1)}_{1bit}$, which contains all quantized $\hat{h}^{(l-1)}_{1bit}$ is broadcast to the corresponding GPU via *Communicator*. Besides, *Communicator* also sends error-compensated information such as scale (the ratio between $\max(\boldsymbol{h}^{(l-1)}) - \min(\boldsymbol{h}^{(l-1)})$ and B) and zeropoint $\min(\boldsymbol{h}^{(l-1)})$ along with the quantized data, to better recover data unbiasedly. As shown in Table 3, the error-compensated data have much smaller size compared with the main data of communicated embeddings or gradients, incurring negligible communication overhead.

Dequantizer. Managed by *Communicator*, each partition itself receives the compressed features $\hat{R}_{1bit}^{(l-1)}$ along with error-compensated information from other partitions. Then

Dequantizer dequantizes them back to 32-bit full precision floating-point values $\tilde{R}^{(l-1)}$, with the help of error-compensated information, to ensure the consistency between the recovered features and the original ones as much as possible. The dequantization process of each embedding $\hat{h}_{1bit}^{(l-1)}$ in $\hat{R}_{1bit}^{(l-1)}$ is expressed as:

$$\tilde{\boldsymbol{h}}^{(l-1)} = \frac{(\max(\boldsymbol{h}^{(l-1)}) - \min(\boldsymbol{h}^{(l-1)}))\hat{\boldsymbol{h}}_{1bit}^{(l-1)}}{B} + \min(\boldsymbol{h}^{(l-1)})$$
(5)

where the recovered $\tilde{m{R}}^{(l-1)}$ includes all dequantized $\tilde{m{h}}^{(l-1)}$.

Trainer. Using $\tilde{\mathbf{R}}^{(l-1)}$ together with $\mathbf{H}^{(l-1)}$ which is already stored in this partition, node embeddings $\mathbf{H}^{(l)}$ of l-th layer are updated according to Equ.1 and 2 by *Trainer*.

The backward pass shares a similar training process as above, except that the communicated data are feature gradients instead, and *Trainer* gradually computes the gradients after the received data are dequantized. Fig.4(b) depicts the workflow of Sylvie-S in each partition as described above. It vastly cuts down the communication overhead compared with the conventional distributed training shown in Fig.4(a). For Sylvie-S, computation in each layer follows the communication step, and the same process occurs on other partitions.

3.3 Asynchronous Curtailed Communication with Computation

To further reduce the epoch training time and cut down the overhead brought by quantization and dequantization operations, inspired by Wan et al. (2022b), we propose Sylvie-A by pipelining the *Low-bit Module* and communication with GNN computation on top of Sylvie. Fig.4 (c) depicts the detailed workflow of Sylvie-A. Different from the vanilla distributed training and Sylvie-S, Sylvie-A directly begins each layer's computation with existing information in this partition. Meanwhile, *Low-bit Module* concurrently quantizes data and communicates the compressed data. Thus, Sylvie-A breaks the sequential order between computation and communication and realizes distributed GNN training in an asynchronous way. Note that in Sylvie-A, the currently overlapped communication will be used in the next epoch, ensuring the data integrity when the computation starts.

Bounded Staleness Adaptor. Asynchronization could introduce stale embeddings and feature gradients, which may affect the model convergence. Although the model convergence is robust against staleness for most datasets or models as shown in §4.2, it is still important to mitigate the negative effects on the convergence in other cases. Sylvie-A implements a simple yet effective *Bounded Staleness Adaptor* to perform compulsory synchronization of embeddings or gradients periodically to control the errors brought by staleness. Though the bounded staleness has already been investigated

in traditional distributed ML works (Xian et al., 2021) to improve SGD convergence, the measures to limit staleness in distributed GNN training have been rare. Specifically, this module conducts one sequential training after a fixed epoch interval ϵ_s to synchronize the latest embeddings or gradients, where ϵ_s can be defined by users as a trade-off between the training throughput and convergence rate. For the rest epochs, the training is still in the pipeline mode. Sylvie-A with *Bounded Staleness Adaptor* enabled are denoted as Sylvie-A ϵ_s . Experiment results in Fig.8 demonstrate that *Bounded Staleness Adaptor* well improves the convergence rate when using the asynchronous pipeline technique.

3.4 GNN Training with Sylvie

We elaborate how Sylvie-S works on GNNs in the forward and backward passes.

Forward Propogation. The forward pass of Sylvie-S in each partition is outlined in Alg.1. The initial node embeddings are set as the feature vectors (line 2). First, we obtain HALO node set so that nodes will be delivered correctly later. Within T epochs, in each GNN layer, a vertex updates its embedding with its one-hop neighbors' embeddings of the last layer, which may include HALO nodes whose embeddings reside in different partitions. Those embeddings are quantized by *Quantizer* in their own partitions (line 10), distributed through network communications (line 11), and then dequantized by *Dequantizer* in the current partition (line 12). The recovered embeddings, together with those originally stored in this GPU, are concatenated into $ilde{m{H}}_n^{(l-1)}$ to update the embeddings. Line 15 in Alg.1 is the matrix format of Equ.1 and 2, where $\boldsymbol{A}_n = \boldsymbol{D}^{-\frac{1}{2}}(\boldsymbol{A}_n + \boldsymbol{I})\boldsymbol{D}^{-\frac{1}{2}}$ is the normalized adjacency matrix, D is the degree matrix, and I is the identity matrix.

Backward Propogation. Alg.2 shows the backward pass of Sylvie-S. In the forward phase, the feature embeddings used for update contain those already stored and communicated ones from other partitions. Therefore, in the backward pass, the feature gradients computed locally is incomplete since the dataflow of more than 1-hop cannot be obtained locally. Therefore, from the L-th to the 1-st layer, the feature gradients $J_n^{(l)}$ also need to be communicated between partitions. Similar as the forward phase, *Quantizer* and *Dequantizer* are in charge of quantizing and dequantizing feature gradients (lines 10 and 12). After integrating the gradients (line 13), the weights can be correctly computed (lines 17).

3.5 Theoretical Analysis

In this section, we study the convergence of Sylvie. We theoretically analyze the unbiased characteristic of *Low-bit Module* and the limited introduced noise, so the model accu-

Algorithm 1 Forward phase of Sylvie-S

```
adjacency matrix A_n, epoch number T, layer size L, local
        node set V_n, weights \mathbf{W}^{(l-1)}
  1: Partition n = 1, 2, ..., N in Parallel:
 2: H_n^{(0)} = X_n \triangleright Initialize node embeddings
3: V_{\text{HALO},n} = \{ \text{node } v \in \mathcal{G}_n : v \notin V_n \}
  2: H_n^{(0)} = X_n
  4: Distribute V_{\text{HALO},n} and receive [V_{\text{HALO},1},...,V_{\text{HALO},N}]
  5: Distribute V_n and receive [V_1, ..., V_N]
  6: [S_1,...,S_N] = [V_n \cap V_{\text{HALO},1},...,V_n \cap V_{\text{HALO},N}]
7: [R_1,...,R_N] = [V_{\text{HALO},n} \cap V_1,...,V_{\text{HALO},n} \cap V_N]
  8:
       for t from 1 to T do
            for l from 1 to L do
  9:
                  \hat{\boldsymbol{S}}_{1bit}^{(l-1)} = \text{quantize}([\boldsymbol{H}_n^{(l-1)}(S_1),...,\boldsymbol{H}_n^{(l-1)}(S_N)])
10:
                 Send \hat{S}_{1bit}^{(l-1)} to partition 1,...,N. Receive \hat{R}_{1bit}^{(l-1)} from partition 1,...,N
11:
                  \hat{m{R}}^{(l-1)} = 	ext{dequantize}(\hat{m{R}}^{(l-1)}_{1bit}) \ \hat{m{H}}^{(l-1)}_n = 	ext{concatenate}(m{H}^{(l-1)}_n, \tilde{m{R}}^{(l-1)})
12:
13:
14:
                      \boldsymbol{H}_{n}^{(l)} = \sigma \left( \boldsymbol{A}_{n}^{\top} \tilde{\boldsymbol{H}}_{n}^{(l-1)} \boldsymbol{W}_{t-1}^{(l)} \right)
15:
                                                                                                           \triangleright Update
                      embeddings
                 else \boldsymbol{H}_n^{(l)} = \boldsymbol{A}_n^{\top} \tilde{\boldsymbol{H}}_n^{(l-1)} \boldsymbol{W}_{t-1}^{(l)} end if
16:
17:
18:
             end for
19:
             \mathcal{L} = \text{Loss}(\text{softmax}(\boldsymbol{H}_n^{(L)}), Y_n)
20:
```

input Partition id n, subgraph \mathcal{G}_n , feature matrix X_n , label Y_n ,

racy and convergence on Sylvie could be well maintained.

21: **end for**

Unbiased Low-bit Module. The following theorems characterize the unbiased feature of quantization, and are referenced from ActNN (Chen et al., 2021).

Theorem 1 (Unbiased embeddings) Assume that $\bar{\mathbf{h}}^{(l)} - \lfloor \bar{\mathbf{h}}^{(l)} \rfloor \sim \mathcal{U}(0,1)$, D is the hidden size of GNN layers, the quantized and dequantized embeddings are unbiased.

$$\mathbb{E}\left[\tilde{\boldsymbol{h}}^{(l)}\right] = \mathbb{E}[\text{Dequantize}(\text{Quantize}(\boldsymbol{h}^{(l)}))] = \boldsymbol{h}^{(l)} \tag{6}$$

$$\operatorname{Var}(\tilde{\boldsymbol{h}}^{(l)}) = \frac{D\left[\max(\boldsymbol{h}^{(l)}) - \min(\boldsymbol{h}^{(l)})\right]^2}{6B^2}$$
(7)

Theorem 2 (Unbiased gradients) There exist random quantization strategies $\hat{\mathbf{C}}$ that give unbiased weight gradients.

$$\mathbb{E}\left[\hat{\boldsymbol{G}}^{(l)}\right] = \boldsymbol{G}^{(l)}$$

$$\operatorname{Var}\left[\hat{\boldsymbol{G}}^{(l)}\right] = \operatorname{Var}\left[\boldsymbol{G}^{(l)}\right] +$$

$$\sum_{m=l}^{L} \mathbb{E}\left[\operatorname{Var}\left[\mathbf{B}^{(l\sim m)}\left(\hat{\boldsymbol{J}}^{(m)}, \hat{\mathbf{C}}^{(m)}\right) \mid \hat{\boldsymbol{J}}^{(m)}\right]\right]$$
 (9)

where $\mathbf{B}^{(l)}$ is the backward function of the l-th layer.

The two theorems prove the quantization and dequantization operations are unbiased, so are the subsequently calculated

Algorithm 2 Backward phase of Sylvie-S

```
input Partition id n, label Y_n, adjacency matrix A_n, epoch num-
           ber T, layer size L, local node set V_n, weights \boldsymbol{W}^{(l-1)}
   1: Partition n = 1, 2, ..., N in Parallel:
           for t from 1 to T do
                  for l from L to 1 do
  3:
                         \begin{aligned} & \textbf{if } l = L \textbf{ then} \\ & \boldsymbol{J}_n^{(L)} = \nabla_{\boldsymbol{H}_n^{(L)}} \mathcal{L} \end{aligned} 
  4:
  5:
  6:
                        oldsymbol{G}_{n}^{(l)} = \left[oldsymbol{A}_{n}	ilde{oldsymbol{H}}_{n}^{(l-1)}
ight]^{	op} \left(oldsymbol{J}_{n}^{(l)} \circ \sigma'\left(oldsymbol{A}_{n}	ilde{oldsymbol{H}}_{n}^{(l-1)}oldsymbol{W}_{t-1}^{(l)}
ight)
ight)
  7:
                         Compute weight gradients
  8:
                         if l > 1 then
                              \begin{aligned} & \boldsymbol{J}_{n}^{(l-1)} = \boldsymbol{A}_{n}^{\top} \left(\boldsymbol{J}_{n}^{(l)} \circ \sigma' \left(\boldsymbol{A}_{n} \tilde{\boldsymbol{H}}_{n}^{(l-1)} \boldsymbol{W}_{t-1}^{(l)}\right)\right) \left[\boldsymbol{W}_{t-1}^{(l)}\right]^{\top} \\ & \triangleright \textit{Compute feature gradients} \\ & \hat{\boldsymbol{S}}_{1bit}^{(l-1)} = \text{quantize}([\boldsymbol{J}_{n}^{(l-1)}(R_{1}),...,\boldsymbol{J}_{n}^{(l-1)}(R_{N})]) \\ & \text{Send } \hat{\boldsymbol{S}}_{1bit}^{(l-1)} \text{ to partition } 1,...,N. \text{ Receive } \hat{\boldsymbol{R}}_{1bit}^{(l-1)} \end{aligned}
  9:
10:
11:
                                from partition 1, ..., N
                               \begin{split} &\hat{\boldsymbol{R}}^{(l-1)} = \text{dequantize}(\hat{\boldsymbol{R}}_{1bit}^{(l-1)}) \\ &\boldsymbol{J}_{n}^{(l-1)} = \boldsymbol{J}_{n}^{(l-1)} + \hat{\boldsymbol{R}}^{(l-1)} \end{split}
 12:
13:
                                                                                                                             ⊳Integrate feature
                               gradients
 14:
                         end if
 15:
                   end for
 16:
                  G = AllReduce(G_n)
                                                                                                      ⊳Gradient synchronization
                   \boldsymbol{W}_t = \boldsymbol{W}_{t-1} - \eta \boldsymbol{G}
 17:
                                                                                                                                    \triangleright Update\ model
 18: end for
```

weights and gradients. Besides, the noise introduced by these two operations in Sylvie is limited, so the model quality can be well kept. Equ.7 and 9 reveal that quantization brings some extra noise to the training data, and the noise is inversely proportional to the number of bits. Though the noise will be aggregated layer-by-layer and result in a paramount accuracy drop (usually >5%) to typical CNNs (Chen et al., 2021), we find its influence on GNNs is negligible. The layer size of GNNs (usually 2 to 4) is far smaller than that of CNNs, making GNNs more noise-tolerant than CNNs (Liu et al., 2021). Therefore, quantization is suitable to be applied to facilitate distributed GNN training. Table 2 also shows the effect of the introduced noise on accuracy can be neglected in practice.

Model Convergence. Given the unbiased gradients, we can establish the convergence of Sylvie. Suppose we have the common SGD in the form of $W_{t+1} = W_t - \eta \hat{G}$, starting from the initial weights W_1 . We make the following assumptions:

- The loss \mathcal{L} is continuous differentiable and $\nabla \mathcal{L}(\boldsymbol{W})$ is β -Lipschitz continuous (Lip, 2022).
- \mathcal{L} is bounded below by \mathcal{L}_{inf} .
- There exists $\sigma^2 > 0$, such that $\forall W, \operatorname{Var}\left[\hat{G}\right] \leq \sigma^2$, where for any vector $\mathbf{x}, \operatorname{Var}[\mathbf{x}] := \mathbb{E}\|\mathbf{x}\|^2 \|\mathbb{E}[\mathbf{x}]\|^2$.

Then we can have the following convergence theorem, taken from Theorem 4.8 in Bottou et al. (2018).

Theorem 3 (Convergence) If $0 < \eta \le \frac{1}{\beta}$, for iteration t in

 $\{1,...,T\}$, where T is the maximum number of iterations, we have

$$\mathbb{E} \left\| \nabla \mathcal{L} \left(\mathbf{W}_{t} \right) \right\|^{2} \leq \frac{2 \left(\mathcal{L} \left(\mathbf{W}_{1} \right) - \mathcal{L}_{inf} \right)}{\eta T} + \eta \beta \sigma^{2} \quad (10)$$

The first term of Equ.10 converges to zero as the number of iterations T goes to infinity. Therefore, the algorithm converges to the neighborhood of a stationary point, where the radius is controlled by the gradient variance.

In addition, different from most compression methods (Chen et al., 2021; Liu et al., 2021), in each epoch, Sylvie only applies quantization to a portion of embeddings and gradients (i.e. those needed for communication), allowing for unbiased gradients to flow through weights. This controls the amount of noise so that the extreme 1-bit compression can be applied with limited accuracy loss. Similar techniques can be found in existing works (Dong et al., 2017; Stock et al., 2020; Tailor et al., 2021) which adopt the subset quantization.

4 EXPERIMENTS

We first compare Sylvie with other distributed full-graph GNN training methods in both the multi-node and single-node settings (§4.1). Then we present the convergence of Sylvie on different datasets and models (§4.2). To explore how quantization affects the performance, we evaluate the accuracy and training time using different bit-widths for quantization (§4.3). Finally, we analyze the overhead of Sylvie (§4.4).

Datasets and Models. We evaluate Sylvie on four real-world large-scale graph benchmarks: Reddit (Hamilton et al., 2017), Yelp (Zeng et al., 2020), Ogbn-products (Hu et al., 2020) and Amazon (He & McAuley, 2016). We choose three popular GNN models, which are commonly adopted in evaluating GNN training: vanilla GCN (Kipf & Welling, 2016), GraphSAGE (Hamilton et al., 2017) and GAT (Veličković et al., 2018).

Baselines. For the baselines, we compare Sylvie with four SOTA distributed full-graph training methods: (1) DGL (Wang et al., 2020): the standard distributed GNN training on top of the latest DGL 0.9; (2) SAR (Mostafa, 2022); (3) PipeGCN (Wan et al., 2022b); (4) BNS-GCN (Wan et al., 2022a): the p value is set to 0.1 as suggested by the paper. p=0 is not practical since it suffers from the worst test accuracy, the slowest convergence and severe overfitting. Baselines are orthogonal to each other in distributed GNN system designs so that we can make a fair comparison.

Testbeds. We implement Sylvie atop the latest stable version of popular GNN training library DGL 0.9 (Wang et al., 2020) with PyTorch 1.10 (Paszke et al., 2019). For all experiments, we use machines equipped with 8 GPUs (NVIDIA

RTX 3090, each has 24GB GDDR6X Memory), one 32-thread CPU (AMD Threadripper PRO 3955WX) and 192GB DDR4 Memory. The intra-server communication (CPU-GPU and GPU-GPU) is based on PCIe 4.0 lanes.

4.1 Performance Evaluation

Performance on Multiple Servers. Training models on multiple machines is becoming a regular necessity nowadays. Fig.5 describes the throughput comparisons between Sylvie and SOTA baselines on three models and four datasets over two machines. Here throughput is defined as the number of epochs run per second, and we normalize the throughput of each method on base of DGL. In each training task, we treat the first 10 epochs as the warmup stage and only record statistics afterward. We can clearly see that Sylvie substantially outperforms other methods by a large margin on each dataset and model. Specifically, Sylvie-S achieves a marvelous throughput improvement of 12.2~21.6× over DGL and far exceeds SAR and PipeGCN. Among the baselines, SAR shows the lowest throughput since it does not cope with the communication overhead, and its computation burden even increases due to the rematerialization. Sylvie-S also delivers 1.4~2.3× larger throughput than BNS-GCN.

Sylvie-A further improves the training performance, which reaches $15.2 \sim 28.1 \times$ speedup over DGL and $1.8 \sim 3.3 \times$ speedup over BNS-GCN. In addition, we note that PipeGCN shows similar performance with DGL because in the multiserver training, the communication cost is immensely larger than computation. In this case, the communication could hardly be hidden so the performance gain is negligible.

We show the detailed normalized training throughput and test accuracy in Table 2. In the multi-server setting, Sylvie always achieves far better training throughput than other methods, demonstrating its effectiveness in large-scale distributed training. Additionally, the test accuracy of Sylvie only suffers from minor degradation. In contrast, BNS-GCN incurs significant accuracy loss of up to 4.9% compared to DGL on GAT, showing its limited generality to other models. The model accuracy of Sylvie-A experiences a moderate fluctuation between $-1.24\% \sim +1.25\%$, due to the stale communicated embeddings & gradients.

Moreover, Fig.6 presents the training throughput of Sylvie in the two-server and three-server systems, respectively. We can observe that Sylvie maintains the great performance and even achieves higher throughput acceleration ratio when the number of servers increases. On both settings, Sylvie offers the best training speedup compared with other methods, while SAR and PipeGCN show very limited performance in large-scale training. In a nutshell, Sylvie can deliver desired performance for larger-scale training scenarios.

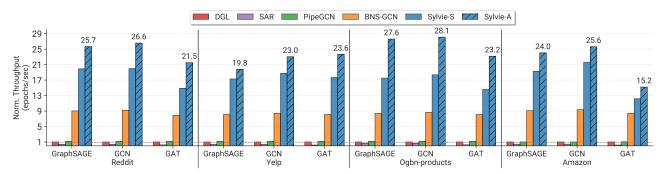


Figure 5. Training throughput of different methods (normalized to that of DGL, shown in the dashed line) when training three representative models on four datasets on two servers. Sylvie outperforms DGL by up to 28.1×.

Table 2. Detailed comparison of training throughput and test accuracy between Sylvie and other baselines when training on two machines, where the best performance is highlighted in bold. Sylvie always outperforms others in throughput on all the models and datasets.

		Reddit		Yelp		Ogbn-products		Amazon	
Model	Method	Thr.	Test Acc.(%)	Thr.	F1-micro(%)	Thr.	Test Acc.(%)	Thr.	Test Acc.(%)
GraphSAGE	DGL	1.00×	97.10 ±0.01	1.00×	65.07±0.19	1.00×	79.19 ±0.15	1.00×	81.29 ±0.02
	SAR	$0.42 \times$	96.02 ± 0.12	$0.37 \times$	60.51 ± 0.09	$0.64 \times$	74.42 ± 0.07	$0.43 \times$	78.85 ± 0.07
	PipeGCN	$1.15 \times$	97.02 ± 0.11	$1.15 \times$	65.14 ± 0.08	$1.19 \times$	79.29 ±0.05	$1.05 \times$	81.27 ± 0.08
	BNS-GCN	$9.02 \times$	97.14 ± 0.01	$8.11 \times$	65.22 ± 0.23	$8.38 \times$	79.11 ± 0.11	$9.08 \times$	80.90 ± 0.05
	Sylvie-S	$19.90 \times$	97.15 ±0.11	$17.29 \times$	65.07 ± 0.23	$17.51 \times$	78.86 ± 0.17	$19.27 \times$	81.22 ± 0.07
	Sylvie-A	$25.66 \times$	96.87 ± 0.03	19.76 ×	64.92 ± 0.38	27.59 ×	$78.85 {\pm} 0.56$	$\textbf{24.01} \times$	81.24 ± 0.11
	DGL	1.00×	94.84±0.58	1.00×	47.50±0.07	1.00×	74.58 ±0.20	1.00×	56.59 ±0.11
	SAR	$0.42 \times$	95.34 ± 0.17	$0.38 \times$	47.00 ± 0.12	$0.65 \times$	70.13 ± 0.10	$0.43 \times$	53.08 ± 0.07
CON	PipeGCN	$1.15 \times$	94.69 ± 0.56	1.16×	47.16 ± 0.01	$1.20 \times$	74.04 ± 0.23	$1.01 \times$	56.56 ± 0.34
GCN	BNS-GCN	$9.18 \times$	95.00 ± 0.33	$8.40 \times$	47.27 ± 0.37	$8.64 \times$	73.54 ± 0.42	$9.34 \times$	56.47 ± 0.60
	Sylvie-S	$19.97 \times$	95.49 ±0.04	$18.76 \times$	48.77 ± 0.14	$18.35 \times$	74.14 ± 0.49	$21.56 \times$	56.07 ± 0.07
	Sylvie-A	$26.56\times$	95.31 ± 0.01	$\textbf{23.02} \times$	47.62 ± 0.30	$\textbf{28.09} \times$	73.78 ± 0.19	$25.60\times$	55.84 ± 0.21
GAT	DGL	1.00×	93.97±0.60	1.00×	44.39±0.16	1.00×	78.14 ±0.12	1.00×	42.84±0.96
	SAR	$0.25 \times$	91.47 ± 0.08	$0.21 \times$	44.30 ± 0.11	$0.27 \times$	76.40 ± 0.06	$0.21 \times$	42.48 ± 0.07
	PipeGCN	$1.14 \times$	94.49 ± 0.64	1.15×	43.75 ± 0.23	1.19×	77.03 ± 0.11	$1.04 \times$	42.37 ± 0.07
	BNS-GCN	$7.86 \times$	89.08 ± 0.63	$8.11 \times$	43.66 ± 0.24	$8.08 \times$	74.07 ± 0.92	$8.43 \times$	40.67 ± 0.79
	Sylvie-S	$14.86 \times$	94.55 ±0.77	17.66×	44.44 ±0.62	$14.57 \times$	78.00 ± 0.01	$12.18 \times$	42.95 ±0.16
	Sylvie-A	21.49×	93.40 ± 0.62	23.62×	43.15 ± 0.63	23.16×	78.38 ± 0.18	15.20×	41.83 ± 0.25

Performance on Single Server. We also test the performance of Sylvie on one single machine with 8 GPUs. Fig.7 shows partial results of the throughput for different methods due to the page limit. Sylvie still outperforms other methods in training throughput, with a maximum of $7.0\times$ speedup over DGL on Ogbn-products when training GraphSAGE. The speedup is relatively less significant compared to the multi-server setting, which involves more partitions with larger communication overhead. This indicates Sylvie is more effective when the training scale is larger.

Communication Volume and Time. To demonstrate the training speedup is due to the reduced communication, we record the actual communication volume per epoch and training time breakdown in Table 3. We observe that Sylvie cuts down the communication volume dramatically. For example, there are originally 5632.6 MB communication per epoch for the Amazon dataset. After deploying Sylvie, there

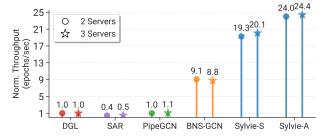


Figure 6. Normalized training throughput on two servers and three servers for GraphSAGE on Amazon.

are only 176.1 MB communicated embeddings & gradients, reducing almost $32\times$ communication volume. Accordingly, the communication time is vastly shortened. The time breakdown shows the communication occupies a very large portion in epoch time, so the training throughput is also improved. Note that we also transmit some error-compensated information, which accounts for much smaller portions (e.g.,

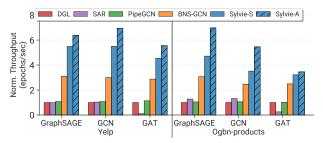


Figure 7. Some throughput results when training on a single machine with 8 GPUs. Sylvie still outperforms others.

Table 3. Epoch communication volume and time breakdown of training GraphSAGE on four datasets over two servers. The communication volume decrease of Sylvie is almost 32×.

	Method	Comn	n. Volume(MB)	Time per Epoch (s)		
	Memod	Main Data Error-compensated		Total	Comm.	
Reddit	DGL	2791.7	0	7.28	6.62	
	Sylvie-S	87.3	10.7	0.37	0.29	
Yelp	DGL	2348.1	0	4.98	4.70	
	Sylvie-S	73.4	4.5	0.29	0.21	
Ogbn-products	DGL	3420.6	0	6.03	5.87	
	Sylvie-S	106.9	26.6	0.34	0.22	
Amazon	DGL	5632.6	0	13.33	11.47	
	Sylvie-S	176.1	11	0.69	0.57	

11.0 MB for Amazon) and incurs negligible overhead.

4.2 Impact on Convergence

We examine the convergence curves of Sylvie in Fig.8. We can see the curves of Sylvie-S are almost identical to that of DGL in this case. However, Sylvie-A gives slower convergence rate at an early phase of some datasets or models. To mitigate the errors by stale embeddings & feature gradients, we train Sylvie-A with *Bounded Staleness Adaptor* when $\epsilon_s = \{2, 5\}$ on Reddit. We observe that with periodical synchronization of the latest data, Sylvie-A boosts the convergence speed well, and the convergence curve approximates that of vanilla training more when the synchronization is more frequent (ϵ_s gets smaller).

4.3 Quantization Effect Analysis

Impact of the Bit-width. To explore how the quantization bit-width *b* contributes to the training throughput improvement, we showcase the training epoch time along with its breakdown for GraphSAGE on Sylvie-S with different quantization bit-widths in Fig.9. As the bit-width goes down, there occurs semi-linear reduction in the communication overhead, thus leading to a speedup on the epoch time. The vanilla method uses FP32 datatype for communication, which occupies nearly the whole epoch time (0.99s out of 1.21s). Switching to FP16 datatype, the communication cost almost decreases by half. When using 1-bit quantization, we cut down almost 89.8% communication overhead and 84.2% training time per epoch compared to the vanilla method.

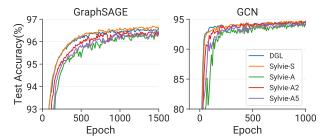


Figure 8. Test accuracy convergence during the training process with DGL, Sylvie-S, Sylvie-A and Sylvie-A with Bounded Staleness Adaptor ($\epsilon_s = \{2, 5\}$) on Reddit.

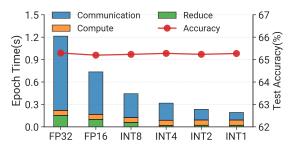


Figure 9. Training time per epoch, its breakdown and corresponding test accuracy when using different quantization bit-widths to train GraphSAGE on Yelp over single server (N=8).

Impact of the Quantization Ratio. To further prove the effectiveness of our method on the model performance, we conduct training by quantizing all the embeddings & feature gradients to 1-bit. Table 4 shows the comparisons of test accuracy between Sylvie-S and quantizing all data. On all datasets and models, quantizing all data will bring serious accuracy loss, e.g. 97.2% in Sylvie-S versus 70.6% in quantizing all. The accuracy drop is due to the overmuch distortion on data by absolute quantization, deteriorating both the forward and backward pass in training.

4.4 Overhead Analysis

Quantization Overhead. To explore how much overhead the *Low-bit module* takes up and find the latent optimization opportunities, we record the time spent on each part in an epoch with Sylvie-S on single server and two servers in Fig.10. Both cases demonstrate that the time consumed by this module occupies the smallest portions, indicating the negligible overhead brought by our methods. Specifically, under the two-server setting, the total ratio of them (5.9%) are even smaller than that of the all-reduce (6.7%). And the communication overhead becomes more dominant in the multi-server setting.

5 FUTURE WORK

Error Compensation. Inevitably, quantization and dequantization process will introduce errors into the training process. These errors of embeddings and embedding gradients will even accumulate in the subsequent layers, seriously af-

Table 4. Test accuracy comparison when training with Sylvie and quantizing all embeddings & feature gradients.

	Reddit		Yelp		Ogbn-products		
	Sylvie-S	Quant All	Sylvie-S	Quant All	Sylvie-S	Quant All	
GraphSAGE	97.15	70.63	65.07	57.14	78.86	58.25	
GCN	95.49	92.60	48.77	42.24	74.14	61.25	
GAT	94.55	91.72	44.44	25.76	78.00	69.95	

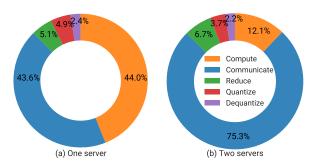


Figure 10. Ratios of different components in epoch time when training GraphSAGE with Sylvie-S on Reddit over single server and two servers. Low-bit Module has negligible overhead.

fecting the model quality when GNN model becomes deep. In our future work, we can utilize two observations to adjust the trade-off between accuracy and training throughput. The first observation is early training epochs can use lower bit-width quantization while latter rounds with higher bit-width without affecting convergence. In this way, we can automatically adjust the bit-width according to the status of training process. Another observation comes from the distribution property of nodes. In GNN layers, the aggregation phase is the source of substantial error, especially at nodes with higher degrees. As the degree of nodes increases, the variance of aggregation values will also increase. Therefore, we can consider applying different bit-width quantization in the future to different nodes according to their importance.

Memory Footprint. Besides offering substantial improvements on the training throughput, the possibility of reducing the memory footprint by quantization is also worth exploring. Using low-precision values is expected to reduce the maximum memory allocated. For instance, when training GraphSAGE on the Ogbn-products dataset, the maximum memory used is about 6882 MB for full-precision FP32 communication and 6642 MB for half-precision FP16 communication. However, because we conduct 1-bit quantization and dequantization using GPUs, we find there are some extra memory expenses during training. The maximum memory used by Sylvie-S reaches 6918 MB, which is slightly higher than vanilla training. Nevertheless, using quantization to reduce memory footprints is still a promising direction and we leave this as a potential future work.

6 CONCLUSION

This work proposes Sylvie, an efficient distributed GNN training framework that enormously reduces the communication cost by quantizing the communicated data to low bit-width values while maintaining the model quality as much as possible. We also integrate Sylvie with the asynchronous pipeline technique and *Bounded Staleness Adaptor* to further enhance the training performance. Extensive experiments show that Sylvie can substantially boost the training throughput by up to $28.1 \times$.

REFERENCES

Lipschitz continuity. https://en.wikipedia.org/wiki/Lipschitz_continuity, 2022.

Nvidia cuda sparse matrix library. https://docs.nvidia.com/cuda/cusparse/index.html#abstract, 2022.

Abadal, S., Jain, A., Guirado, R., López-Alonso, J., and Alarcón, E. Computing graph neural networks: A survey from algorithms to accelerators. *CoRR*, abs/2010.00130, 2021.

Alistarh, D., Grubic, D., Li, J., Tomioka, R., and Vojnovic, M. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In *Advances in Neural Information Processing Systems*, NeurIPS '17, 2017.

Bottou, L., Curtis, F. E., and Nocedal, J. Optimization methods for large-scale machine learning. *CoRR*, abs/1606.04838, 2018.

Chen, J., Ma, T., and Xiao, C. Fastgen: Fast learning with graph convolutional networks via importance sampling. In *International Conference on Learning Representations*, ICLR '18, 2018.

Chen, J., Zheng, L., Yao, Z., Wang, D., Stoica, I., Mahoney, M., and Gonzalez, J. Actnn: Reducing training memory footprint via 2-bit activation compressed training. In Proceedings of the 38th International Conference on Machine Learning, ICML '21, 2021.

Chiang, W.-L., Liu, X., Si, S., Li, Y., Bengio, S., and 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 '19, 2019.

Courbariaux, M., Bengio, Y., and David, J.-P. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems*, NeurIPS '15, 2015.

- Dong, Y., Ni, R., Li, J., Chen, Y., Zhu, J., and Su, H. Learning accurate low-bit deep neural networks with stochastic quantization. *CoRR*, abs/1708.01001, 2017.
- Feng, B., Wang, Y., Li, X., Yang, S., Peng, X., and Ding, Y. Sgquant: Squeezing the last bit on graph neural networks with specialized quantization. In 2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI), 2020.
- Fey, M. and Lenssen, J. E. Fast graph representation learning with pytorch geometric. *CoRR*, abs/1903.02428, 2019.
- Hamilton, W., Ying, Z., and Leskovec, J. Inductive representation learning on large graphs. In Advances in Neural Information Processing Systems, NeurIPS '17, 2017.
- He, R. and McAuley, J. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, 2016.
- Hu, W., Fey, M., Zitnik, M., Dong, Y., Ren, H., Liu, B., Catasta, M., and Leskovec, J. Open graph benchmark: Datasets for machine learning on graphs. In *Advances in Neural Information Processing Systems*, NeurIPS '20, 2020.
- Huang, W., Zhang, T., Rong, Y., and Huang, J. Adaptive sampling towards fast graph representation learning. In Proceedings of the 32nd International Conference on Neural Information Processing Systems, NIPS'18, 2018.
- Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., and Kalenichenko, D. Quantization and training of neural networks for efficient integerarithmetic-only inference. In *Proceedings of the IEEE* Conference on Computer Vision and Pattern Recognition, CVPR '18, 2018.
- Jia, Z., Lin, S., Gao, M., Zaharia, M., and Aiken, A. Improving the accuracy, scalability, and performance of graph neural networks with roc. In *Proceedings of Machine Learning and Systems*, MLSys '20, 2020.
- Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, ICLR '16, 2016.
- Krishnamoorthi, R. Quantizing deep convolutional networks for efficient inference: A whitepaper. *CoRR*, abs/1806.08342, 2018.
- Liu, Z., Zhou, K., Yang, F., Li, L., Chen, R., and Hu, X. Exact: Scalable graph neural networks training via extreme activation compression. In *International Conference on Learning Representations*, ICLR '21, 2021.

- Ma, L., Yang, Z., Miao, Y., Xue, J., Wu, M., Zhou, L., and Dai, Y. NeuGraph: Parallel deep neural network computation on large graphs. In 2019 USENIX Annual Technical Conference, USENIX ATC '19, 2019.
- Mostafa, H. Sequential aggregation and rematerialization: Distributed full-batch training of graph neural networks on large graphs. In *Proceedings of Machine Learning and Systems*, MLSys '22, 2022.
- Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J.,
 Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga,
 L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison,
 M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L.,
 Bai, J., and Chintala, S. Pytorch: An imperative style,
 high-performance deep learning library. In Advances in
 Neural Information Processing Systems, NeurIPS '19,
 2019.
- Ramezani, M., Cong, W., Mahdavi, M., Kandemir, M., and Sivasubramaniam, A. Learn locally, correct globally: A distributed algorithm for training graph neural networks. In *International Conference on Learning Representations*, ICLR '22, 2022.
- Stock, P., Fan, A., Graham, B., Grave, E., Gribonval, R., Jegou, H., and Joulin, A. Training with quantization noise for extreme model compression. In *International Conference on Learning Representations*, ICLR '20, 2020.
- Tailor, S. A., Fernandez-Marques, J., and Lane, N. D. Degree-quant: Quantization-aware training for graph neural networks. *CoRR*, abs/2008.05000, 2021.
- Thorpe, J., Qiao, Y., Eyolfson, J., Teng, S., Hu, G., Jia, Z., Wei, J., Vora, K., Netravali, R., Kim, M., and Xu, G. H. 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 '21, 2021.
- Tripathy, A., Yelick, K., and Buluç, A. Reducing communication in graph neural network training. In SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, 2020.
- Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., and Bengio, Y. Graph attention networks. In International Conference on Learning Representations, ICLR '18, 2018.
- Wan, C., Li, Y., Li, A., Kim, N. S., and Lin, Y. Bns-gen: Efficient full-graph training of graph convolutional networks with partition-parallelism and random boundary node sampling. In *Proceedings of Machine Learning and Systems*, MLSys '22, 2022a.

- Wan, C., Li, Y., Wolfe, C. R., Kyrillidis, A., Kim, N. S., and Lin, Y. Pipegcn: Efficient full-graph training of graph convolutional networks with pipelined feature communication. In *International Conference on Learning Representations*, ICLR '22, 2022b.
- Wang, M., Zheng, D., Ye, Z., Gan, Q., Li, M., Song, X., Zhou, J., Ma, C., Yu, L., Gai, Y., Xiao, T., He, T., Karypis, G., Li, J., and Zhang, Z. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *CoRR*, abs/1909.01315, 2020.
- Wang, Y., Feng, B., and Ding, Y. Qgtc: accelerating quantized graph neural networks via gpu tensor core. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, 2022.
- Wen, W., Xu, C., Yan, F., Wu, C., Wang, Y., Chen, Y., and Li, H. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Advances in Neural Information Processing Systems*, NeurIPS '17, 2017.
- Xian, L., Li, B., Liu, J., Guo, Z., and Du, D. H. C. H-ps: A heterogeneous-aware parameter server with distributed neural network training. *IEEE Access*, 9:44049–44058, 2021.
- Zeng, H., Zhou, H., Srivastava, A., Kannan, R., and Prasanna, V. Graphsaint: Graph sampling based inductive learning method. In *International Conference on Learning Representations*, ICLR '20, 2020.
- Zhang, M. and Chen, Y. Link prediction based on graph neural networks. In *Advances in Neural Information Processing Systems*, NeurIPS '18, 2018.
- Zhao, Y., Wang, D., Bates, D., Mullins, R., Jamnik, M., and Lio, P. Learned low precision graph neural networks. *CoRR*, abs/2009.09232, 2020.
- Zheng, D., Ma, C., Wang, M., Zhou, J., Su, Q., Song, X., Gan, Q., Zhang, Z., and Karypis, G. Distdgl: Distributed graph neural network training for billion-scale graphs. *CoRR*, abs/2010.05337, 2021.
- Zhu, R., Zhao, K., Yang, H., Lin, W., Zhou, C., Ai, B., Li, Y., and Zhou, J. Aligraph: a comprehensive graph neural network platform. *Proceedings of the VLDB Endowment*, 12:2094–2105, 2019.
- Zou, D., Hu, Z., Wang, Y., Jiang, S., Sun, Y., and Gu, Q. Layer-dependent importance sampling for training deep and large graph convolutional networks. In *Proceedings* of the 33rd International Conference on Neural Information Processing Systems, NIPS'19, 2019.