Adaptive Message Quantization and Parallelization for Distributed Full-graph GNN Training

Borui Wan[1] **Juntao Zhao**[1] **Chuan Wu**[1]

Abstract

Distributed full-graph training of Graph Neural Networks (GNNs) over large graphs is bandwidth-demanding and time-consuming. Frequent exchanges of node features, embeddings and embedding gradients (all referred to as *messages*) across devices bring significant communication overhead for nodes with remote neighbors on other devices (*marginal nodes*) and unnecessary waiting time for nodes without remote neighbors (*central nodes*) in the graph. This paper proposes an efficient GNN training system, AdaQP, to expedite distributed full-graph GNN training. We stochastically quantize messages transferred across devices to lower-precision integers for communication traffic reduction and advocate communication-computation parallelization between marginal nodes and central nodes. We provide theoretical analysis to prove fast training convergence (at the rate of $O(T^{-1})$ with T being the total number of training epochs) and design an adaptive quantization bit-width assignment scheme for each message based on the analysis, targeting a good trade-off between training convergence and efficiency. Extensive experiments on mainstream graph datasets show that AdaQP substantially improves distributed full-graph training's throughput (up to $3.01\times$) with negligible accuracy drop (at most 0.30%) or even accuracy improvement (up to 0.19%) in most cases, showing significant advantages over the state-of-the-art works. The code is available at https://github.com/raywan-110/AdaQP.

1 Introduction

Graph Neural Networks (GNNs) have received increased attention from the AI community for their superior performance on graph-based tasks such as node classification (Kipf & Welling, 2017), link prediction (Zhang & Chen, 2018) and graph classification (Xu et al., 2019). For each node of a graph, a GNN typically aggregates features or embeddings of the node's neighbors iteratively and then uses them to create the node's own embedding. This process is referred to as *message-passing* (Gilmer et al., 2017), which enables GNNs to learn better representative embeddings from graph structures than traditional graph learning methods (Wu et al., 2020; Zhang et al., 2020).

For a k-layer GNN, the message-passing paradigm requires features and embeddings in the k-hop neighborhood of the training nodes to be retrieved and stored on a device (e.g., GPU) for computation, leading to high memory overhead. When training on large graphs, the memory consumption may easily exceed the memory capacity of a single device, and GNN training with graph sampling has hence been widely studied (Hamilton et al., 2017b; Chen et al., 2018; Chiang et al., 2019; Zeng et al., 2020; Wan et al., 2022a): the large input graphs are partitioned among multiple devices and machines; each worker (device) samples partial neighborhood of its training nodes and fetches features of sampled neighbors from other devices/machines if they are not in the local graph partition. Such graph sampling reduces memory, computation and communication overheads during distributed GNN training, at the cost of indispensable information loss for graph learning, as compared to full-graph training. Besides, sampling introduces extra time overhead due to running (sophisticated) sampling algorithms (Liu et al., 2021a; Kaler et al., 2022).

Unlike sampling-based GNN training, distributed full-graph training allows learning over the complete input graphs, retaining whole graph structure information. Each device requires messages of all 1-hop neighbors of nodes in its graph partition during iterative training, fetching the respective data from devices where they are stored/computed. The need for frequent message exchanges across devices renders the major performance bottleneck for training. Besides, different devices require different numbers of messages from others, which generates irregular all2all communications, leading to communication stragglers in each communication round. Such overheads of distributed full-graph training have also been echoed in recent literature (Wan et al., 2022b;

[1]Department of Computer Science, University of Hong Kong, Hong Kong, China. Correspondence to: Borui Wan <wanborui@connect.hku.hk>.

Proceedings of the 6th MLSys Conference, Miami Beach, FL, USA, 2023. Copyright 2023 by the author(s).

Peng et al., 2022; Cai et al., 2021).

A few studies have investigated different perspectives to improve distributed full-graph training, including graph partition algorithms and memory management (Ma et al., 2019; Jia et al., 2020), communication planing (Cai et al., 2021) and communication-avoiding training with staleness (Thorpe et al., 2021; Wan et al., 2022b; Peng et al., 2022). Nevertheless, none of them considers compressing remote messages to reduce communication traffic for training expedition.[1] Unlike the above methods, message compression can reduce data volumes for both communications between remote devices and data movement from device to host. The latter occurs when messages need to be moved from device memory to host memory first and then transferred to remote devices. (when GPUDirect RDMA is not available in the cluster).

While messages are being exchanged, embedding computation of nodes with all neighbors located locally often waits for the completion of message transfers in synchronous full-graph training (Ma et al., 2019; Jia et al., 2020; Cai et al., 2021), which is not needed. Although existing staleness-based methods eliminate part of the waiting time by pipelining communication with computation (Wan et al., 2022b) or skipping node broadcast and using historical embeddings for computation (Peng et al., 2022), they may lead to slower training convergence (Wan et al., 2022b; Peng et al., 2022), increasing the wall-clock time to achieve the same model accuracy as compared to their synchronous counterparts. Disparate handling of local nodes with and without remote neighbors has not been found in both synchronous and asynchronous full-graph training works in the literature.

We propose AdaQP, an efficient distributed full-graph GNN training system that accelerates GNN training from two perspectives: adaptive quantization of messages and parallelization of computation of central nodes and message transfers of marginal nodes on each device. Our main contributions are summarized as follows:

▷ We apply adaptive stochastic integer quantization to messages dispatched from each device to others, which reduces the numerical precision of messages and hence the size of transferred data. To our best knowledge, we are the first to apply stochastic integer quantization to expedite distributed full-graph training. We provide a theoretical convergence guarantee and show that the convergence rate is still $O(T^{-1})$, identical to that of no-compression training and better than sampling-based (Cong et al., 2020) and staleness-based (Wan et al., 2022b; Peng et al., 2022) GNN training. Using insights from the convergence analysis, an adaptive bit-width assignment scheme is proposed based on a bi-objective optimization, that assigns suitable quantization bit-width to the transferred messages to alleviate unbalanced data volumes from/to different devices and achieve a good trade-off between training convergence and efficiency.

▷ We further decompose the graph partition on each device into a *central graph* and a *marginal graph*, and overlap the computation time of the former with the message communication time of the latter, to maximize training speed and resource utilization. Since the communication overhead always dominates the training process (Sec. 2.2), the computation time of the central graph can be easily hidden within the communication time without introducing any staleness that influences training convergence.

▷ We implement AdaQP on DGL (Wang, 2019) and PyTorch (Paszke et al., 2019), and conduct extensive evaluation. Experimental results show that AdaQP significantly reduces the communication time by 80.94% maximum and 79.98% on average, improves the training throughput by $2.19 \sim 3.01\times$ with acceptable accuracy fluctuations (-0.30% $\sim +0.19\%$), and outperforms state-of-the-art (SOTA) works on distributed full-graph training on most of the mainstream graph datasets.

2 Background and Motivation

2.1 GNN Message Passing

The message passing paradigm of GNN training can be described by two stages, aggregation and update (Gilmer et al., 2017):

$$h^l_{N(v)} = \phi^l(h^{l-1}_u | u \in N(v)) \tag{1}$$

$$h^l_v = \psi^l(h^{l-1}_v, h^l_{N(v)}) \tag{2}$$

Here $N(v)$ denotes the neighbor set of node v. h^l_v is the learned embedding of node v at layer l. ϕ^l is the aggregation function of layer l, which aggregates intermediate node embeddings from $N(v)$ to derive the aggregated neighbor embedding $h^l_{N(v)}$. ψ^l is the update function that combines $h^l_{N(v)}$ and h^{l-1}_v to produce h^l_v.

The two-stage embedding generation can be combined into a weighted summation form, representing the cases in most mainstream GNNs (e.g., GCN (Kipf & Welling, 2017), GraphSAGE (Hamilton et al., 2017a)):

$$h^l_v = \sigma(W^l \cdot (\sum_u^{\{v\}\cup N(v)} \alpha_{u,v} h^{l-1}_u)) \tag{3}$$

where $\alpha_{u,v}$ is the coefficient of embedding h^{l-1}_u, W^l is the weight matrix of layer l and σ is the activation function. We will use this form in our theoretical analysis in Sec. 4.

[1]Model gradient compression has been extensively studied to accelerate distributed DNN training (Alistarh et al., 2017; Yu et al., 2019; Wu et al., 2018). However, for GNNs, the size of model gradients is typically much smaller than those of node features and embeddings (e.g., for the ogbn-products dataset, a three-layer GCN with a hidden size of 256 has 0.55MB model gradients, but 1.17GB features and 3.00GB embeddings), making the transferring of messages much more costly than that of gradients.

2.2 Inefficient Vanilla Distributed Full-graph Training

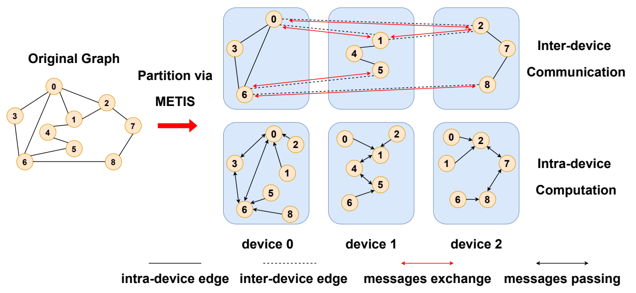


Figure 1. An illustration of vanilla distributed full-graph training. The original graph is partitioned into 3 partitions by METIS (Karypis & Kumar, 1997) and deployed on 3 devices. Inter-device communication and intra-device computation are repeated in the forward (backward) pass of each GNN layer.

Table 1. Communication overhead in Vanilla. xM-yD means the whole graph is partitioned into $x \times y$ parts and dispatched to x servers, each using y available training devices (GPUs).

Dataset	Partition Setting	Communication Cost	Remote Neighbor Ratio
Reddit (Hamilton et al., 2017a)	2M-1D	66.78%	41.54%
	2M-2D	75.20%	62.60%
ogbn-products (Hu et al., 2020)	2M-2D	75.59%	31.09%
	2M-4D	76.67%	40.52%
AmazonProducts (Zeng et al., 2020)	2M-2D	75.58%	39.75%
	2M-4D	78.22%	53.00%

In vanilla distributed full-graph training (Fig. 1, referred to as *Vanilla*), interleaving communication-computation stages exist in each GNN layer during both forward pass and backward pass for generating embeddings or embedding gradients. Therefore, multiple transfers of 1-hop remote neighbors' messages (specifically, transferring features and embeddings in the forward pass and embedding gradients, also denoted as *errors* (Goodfellow et al., 2016), in the backward pass) lead to large communication overhead. To illustrate it, we train a three-layer GCN on representative datasets (all experiments in this section use this GCN, detailed in Sec. 5) and show the communication cost, which is computed by dividing the average communication time by the average per-epoch training time among all devices, in Table 1. We observe that communication time dominates training time. Further, with the increase of partition number, the communication cost becomes larger due to the growth of the remote neighbor ratio (computed by dividing the average number of remote 1-hop neighbors by the average number of nodes among partitions).

Besides, with the mainstream graph partition algorithms (e.g., METIS (Karypis & Kumar, 1997)), the number of nodes whose messages are transferred varies among different device pairs, creating unbalanced all2all communications. This unique communication pattern exists throughout the GNN training process, which does not occur in distributed DNN training (where devices exchange same-size model gradients). Fig. 2 shows the data size transferred across different device pairs in GCN's first layer

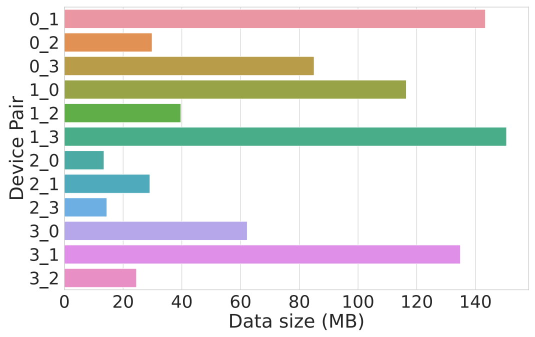


Figure 2. Comparison of data size transferred across each device pair when training the GCN on AmazonProducts with 4 partitions.

Table 2. Computation time of central nodes and transfer time of 2-bit quantized messages of marginal nodes on ogbn-products with 8 partitions.

Device	comm. (s)	Comp. (s)	Device	comm. (s)	Comp. (s)
Device0	0.08	0.04	Device4	0.08	0.06
Device1	0.09	0.05	Device5	0.13	0.06
Device2	0.10	0.05	Device6	0.09	0.06
Device3	0.08	0.05	Device7	0.12	0.05

when training on AmazonProducts, partitioned among 4 devices. There is a significant imbalance among data sizes transferred across different device pairs, which leads to unbalanced communication time across the devices in each communication round, affecting the overall training speed.

Further, Vanilla and previous works (Wan et al., 2022b; Peng et al., 2022; Cai et al., 2021) do not consider that in the forward (backward) pass of each training iteration, the computation of central nodes can directly start without waiting for message exchanges. Overlapping the computation time of central nodes with the communication time of marginal nodes can help further improve the training throughput. As communication renders the major bottleneck in GNN training, we observed that central nodes' computation time can be well hidden within marginal nodes' communication time. In Table 2, we show central nodes' computation time and marginal nodes' communication time when the transferred messages are quantized with a bit-width of 2 (i.e., the numerical precision is 2-bit), rendering the lowest communication volumes. Even under this extremely low communication, communication time is still longer than the central nodes' computation time. When central node computation is hidden within communication, Fig. 3 shows the reduction of model computation time on each device, by 23.20% to 55.44%.

2.3 Stochastic Integer Quantization

As a lossy compression method, stochastic integer quantization (Chen et al., 2021) has been applied to quantize the DNN model for efficient integer arithmetic inference (Zhu et al., 2020; Tailor et al., 2021; Feng et al., 2020), or compress activations to reduce memory footprint during the

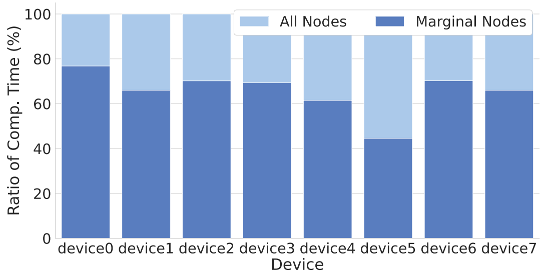


Figure 3. Comparison between computation time of all nodes and computation time of marginal nodes when training on ogbn-products with 8 partitions.

forward pass (Chen et al., 2021; Liu et al., 2021b). Differently, we apply it to reduce communication data volumes in distributed full-graph training. For a given message vector h_v^l of node v in layer l, the b_v-bit quantized version of h_v^l is:

$$\tilde{h}_{v_b}^l = \tilde{q}_b(h_v^l) = round_{st}(\frac{h_v^l - Z_v^l}{S_{v_b}^l}) \tag{4}$$

where $\tilde{q}_b$ denotes stochastic integer quantization operation, $round_{st}(\cdot)$ is the stochastic rounding operation (Chen et al., 2020), $Z_v^l = \min(h_v^l)$ is referred to as the zero-point (the minimum value among elements in vector h_v^l) and $S_{v_b}^l = \frac{max(h_v^l) - min(h_v^l)}{2^{b_v}-1}$ is a scaling factor that maps the original floating-point vector into the integer domain, where b_v is typically chosen among $\{2, 4, 8\}$. A larger quantization bit-width b_v introduces less numerical precision loss, but not as much data size reduction as a smaller value. Received quantized messages are de-quantized into floating-point values for subsequent computation:

$$\hat{h}_v^l = dq_b(\tilde{h}_{v_b}^l) = \tilde{h}_{v_b}^l S_{v_b}^l + Z_v^l \tag{5}$$

where $\hat{h}_v^l$ represents the de-quantized message vector and dq_b denotes de-quantization operation. Following (Chen et al., 2021), $\hat{h}_v^l$ is *unbiased* and *variance bounded*:

Theorem 1. *With stochastic integer quantization and deterministic de-quantization operations $\tilde{q}_b(\cdot)$ and $dq_b(\cdot)$ in Eqn. (4) and Eqn. (5), $\hat{h}_v^l$ is an unbiased and variance bounded estimate of the original input h_v^l, that $\mathbb{E}[\hat{h}_v^l] = \mathbb{E}[dq_b(\tilde{h}_{v_b}^l)] = h_v^l, \mathbb{V}ar[\hat{h}_v^l] = \frac{D_v^l(S_{v_b}^l)^2}{6}$. D_v^l is the dimension of vector h_v^l.*

The good mathematical properties of the quantization method serve as the basis for us to derive a theoretical guarantee of GNN training convergence (Sec. 4).

We propose an efficient system AdaQP, incorporating adaptive message quantization and computation-communication parallelization design, to improve distributed full-graph training efficiency. Fig. 4 gives a high-level illustration of the benefits of AdaQP, as compared to Vanilla.

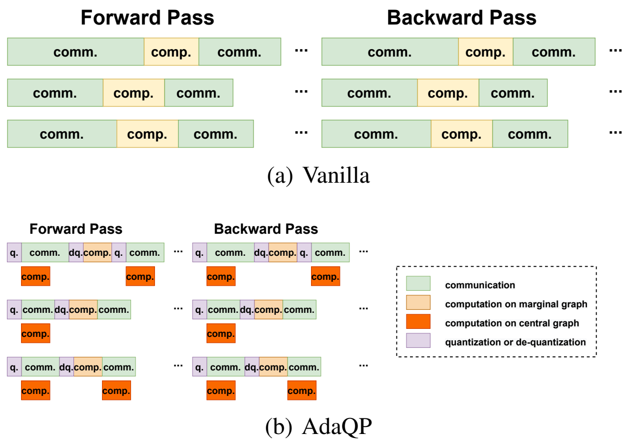

Forward Pass
Backward Pass
comm.
comp.
(a) Vanilla
q.
dq.
communication
computation on marginal graph
computation on central graph
quantization or de-quantization
(b) AdaQP

Figure 4. Comparison of Vanilla and AdaQP.

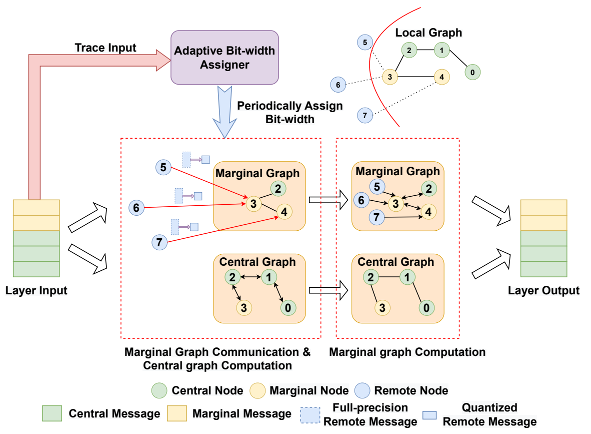

Trace Input
Adaptive Bit-width Assigner
Local Graph
Periodically Assign Bit-width
Marginal Graph
Central Graph
Layer Input
Layer Output
Marginal Graph Communication & Central graph Computation
Marginal graph Computation
Central Node
Marginal Node
Remote Node
Central Message
Marginal Message
Full-precision Remote Message
Quantized Remote Message

Figure 5. Workflow of AdaQP on each device: a per-layer view of GNN training. Only the messages receiving process is illustrated for clarity of the figure.

3 System Design

We study distributed full-graph GNN training using devices (aka workers) on multiple physical machines. The large input graph is partitioned among the devices. Fig. 5 shows the workflow of distributed full-graph training on each device using AdaQP.

3.1 Overview

The graph partition at each device is decomposed into a *central graph*, which contains central nodes and their neighbors, and a *marginal graph*, which includes marginal nodes and their local neighbors. Messages from other devices only need to be passed to the marginal graph for computation. In the forward pass (backward pass) of each GNN layer, for nodes in the marginal graph, quantization is applied to all outgoing messages to other devices, using a bit-width assigned by the Adaptive Bit-Width Assigner, and de-quantization is done on all received messages before the computation stage begins. For central nodes in the central graph, they can enter their computation stage in parallel with the communication for the marginal graph. The Adaptive Bit-width Assigner traces the input data to each GNN layer, and periodically adjusts the bit-width for each message by solving a bi-objective problem with the latest traced data.

3.2 Naive Message Quantization

We perform stochastic integer quantization on the messages transferred across devices during the forward (backward) pass. Consider the aggregation step of GNN layer l for computing embedding of a node v, as given in Eqn. 1 in Sec. 2.1. Let $N_L(v)$ and $N_R(v)$ denote the local and remote neighbor sets of node v. We rewrite the aggregation step as:

$$h^l_{N(v)} = \phi^l(h^{l-1}_u | u \in \{N_L(v) \cup N_R(v)\}) \tag{6}$$

Quantization is only performed on messages in $N_R(v)$ to reduce the communication data size. The benefit brought by naive message quantization is obvious; however, it requires frequently performing quantization and de-quantization operations before and after each device-to-device communication, which adds extra overheads to the training workflow. We notice that quantization (Eqn. 4) and de-quantization (Eqn. 5) themselves are particularly suitable for parallel computing since they just perform the same linear mapping operations to all elements in a message vector. Therefore, we implement the two operations with efficient CUDA kernels. Sec 5.4 demonstrates that compared to training expedition brought by quantization, the extra overheads are almost negligible.

3.3 Dynamic Adaptive Bit-width Assignment

Simply assigning the same bit-width to all messages cannot achieve a good tradeoff between training convergence and training efficiency (Sec. 4); adaptively assigning different bit-widths to different messages is necessary. To support transferring messages quantized with multiple bit-widths between devices, each device first establishes multiple sending buffers for messages of different bit-widths, whose sizes are determined by the Adaptive Bit-width Assigner, and then broadcasts the sizes of the buffers to other devices. Each device also uses received sending buffer sizes from others to set up multiple receiving buffers. The adaptive bit-width assigning process is shown in Fig. 6. Given a bit-width update period, each training device launches an Adaptive Bit-width Assigner which keeps tracing all GNN layers' inputs. The assigner periodically sends traced data from the last period to the master assigner (residing in the device with rank 0) and then blocks the current training worker, waiting for the generation of new bit-width assignment results (step2). At the same time, the master assigner uses gathered data to build a variance-time bi-objective problem, whose variables are bit-widths of the message groups. Since the bit-width assignment results for each GNN layer have no dependence

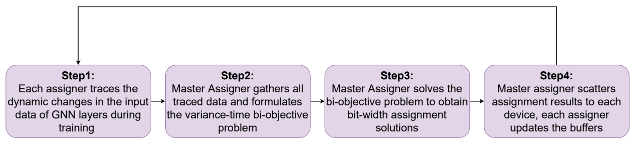

Step1:
Each assigner traces the dynamic changes in the input data of GNN layers during training
Step2:
Master Assigner gathers all traced data and formulates the variance-time bi-objective problem
Step3:
Master Assigner solves the bi-objective problem to obtain bit-width assignment solutions
Step4:
Master assigner scatters assignment results to each device, each assigner updates the buffers

Figure 6. Adaptive bit-width assignment process.

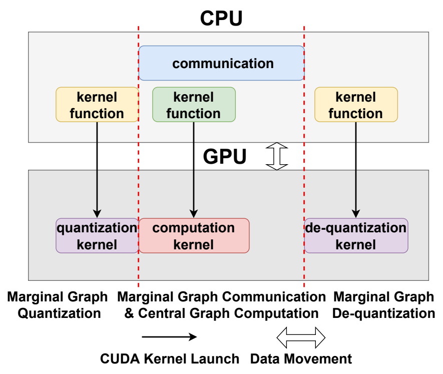


Figure 7. Our GPU resources isolation strategy.

on each other, we create a thread pool in the master device to compute each layer's solution in parallel (step 3). After that, the master assigner scatters the bit-width solutions to corresponding devices, and then each device uses the latest assignment results to update its sending and receiving buffers (step 4).

3.4 Computation-Communication Parallelization

Model computation and message quantization are both compute-intensive. Therefore overlapping computation on the central graph with the quantized message transfers on the marginal graph can lead to an overall slow-down due to the contention for GPU compute resources, which is indicated in previous works (Agarwal et al., 2022). To tackle this issue, we apply a simple yet useful resource isolation strategy based on the observations in Sec. 2.2, that is, the time for transferring extremely quantized messages is still large enough to hide the computation time of the central graph. We control CUDA kernel launching time instead of letting GPUs schedule different CUDA streams freely.

As shown in Fig. 7, we further divide the computation-communication overlapping stage in Fig. 5 into three fine-grained stages, ensuring that in each stage the GPU compute resources are only used by one among quantization, de-quantization and central graph computation. Since communication on the marginal graph only requires the GPU bandwidth, we force the computation CUDA kernel launching and execution on the central graph to be in the second stage with marginal-graph communication, isolating the GPU resource usage between central and marginal graphs. We wrap different types of kernels in GPU with different

CUDA streams and consider the situation that CUDA kernel's launching and execution are asynchronous. We use both CPU and GPU events for synchronization.

4 Convergence Guarantee and Bi-objective Bit-width Assignment

We next show the convergence bound and rate of AdaQP and then formulate a variance-time bi-objective optimization problem for adaptive bit-width assignment.

4.1 Impact of Message Quantization on Training Convergence

We consider widely adopted gradient descent (GD) algorithm (Avriel, 2003) in full-graph GNN training. Similar to previous studies (Fu et al., 2020; Chen et al., 2020), the full-graph training can be modeled as the following non-convex empirical risk minimization problem:

$$\min_{\mathbf{w}_t \in R^D} \mathbb{E}[\mathcal{L}(\mathbf{w}_t)] = \frac{1}{N}\sum_i^N \mathcal{L}_i(\mathbf{w}_t) \quad \mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \tilde{\mathbf{g}}_t \tag{7}$$

where $\mathbf{w}_t$ denotes the parameters at the t-th epoch, D is the dimension of $\mathbf{w}_t$, N denotes the total number of samples (nodes) in the full-graph, and $\mathcal{L}_i(\cdot)$ is the loss on sample i. $\tilde{\mathbf{g}}_t = \nabla \mathcal{L}(\tilde{\mathbf{w}}_t)$ denotes the stochastic gradient and α is the learning step size. Since each training epoch uses full-batch samples, variance in $\tilde{\mathbf{g}}_t$ is only brought by performing stochastic integer quantization on messages. We have Theorem 2 based on standard assumptions for convergence analysis (Fu et al., 2020; Allen-Zhu, 2017).

Assumption 1. *for* $\forall \mathbf{w}_t, \mathbf{w}'_t \in R^D$ *in the t-th epoch:*

A.1 *($\mathcal{L}_2 - Lipschitz$)* $||\nabla \mathcal{L}(\mathbf{w}_t) - \nabla \mathcal{L}(\mathbf{w}'_t)|| \leq L_2 ||\mathbf{w}_t - \mathbf{w}'_t||$*;*

A.2 *(*existence of global minimum*)* $\exists\, \mathcal{L}^*\ s.t.\ \mathcal{L}(\mathbf{w}_t) \geq \mathcal{L}^*$*;*

A.3 *(*unbiased stochastic gradient*)* $\mathbb{E}[\tilde{\mathbf{g}}_t] = \mathbf{g}_t$;

A.4 *(*bounded variance*)* $\mathbb{E}[||\mathbf{g}_t - \tilde{\mathbf{g}}_t||] \leq Q$.

Theorem 2. *Suppose our distributed full-graph GNN training runs for T epochs using a fixed step size $\alpha < \frac{2}{L_2}$. Select t randomly from $\{1, \cdots, T\}$. Under Assumption 1, we have*

$$\mathbb{E}[||\nabla \mathcal{L}(\bar{\mathbf{w}}_t)||^2] \leq \frac{2(\mathcal{L}(\mathbf{w}_1) - \mathcal{L}^*)}{T(2\alpha - \alpha^2 L_2)} + \frac{\alpha L_2 Q^2}{2 - \alpha L_2}. \tag{8}$$

All the detailed proofs can be found in Appendix A. Similar to the convergence result of a standard SGD algorithm (Robbins & Monro, 1951), the bound in Theorem 2 includes two terms, while the second term is totally different from its SGD counterpart, where variance is not introduced due to sampling variance but message quantization. The first term in the bound goes to 0 as $T \to \infty$, which shows an $O(T^{-1})$ convergence rate. AdaQP's convergence rate is the same as that with Vanilla, and faster than those of sampling-based methods ($O(T^{-\frac{1}{2}})$ (Cong et al., 2020)) and staleness-based methods ($O(T^{-\frac{2}{3}})$ (Wan et al., 2022b) or $O(T^{-\frac{1}{2}})$ (Peng et al., 2022)).

During training, the gradient variance exists in the weight matrix in each layer l of an L-layer GNN. Let $\mathbf{w} = \{\mathbf{w}^l\}_{l=1}^L$ denotes the set of weight matrices of GNN, we then show the gradient variance upper bound Q^l for each $\mathbf{w}^l$. Let $\bar{h}_v^{l-1} = \sum_u^{\{v\}\cup N(v)} \alpha_{u,v} h_u^{l-1}$ in Eqn. 3, and $\frac{\partial \bar{\mathcal{L}}}{\partial h_v^l} = \sum_u^{\{v\}\cup N(v)} \alpha_{u,v} \frac{\partial \mathcal{L}}{\partial h_u^l}$ in its backward pass counterpart. Based on Theorem 1, we derive Q^l under Assumption 2:

Assumption 2. *For each layer $l \in \{1, 2, \cdots, L\}$ in the GNN and for each node v in the full-graph, L_2 norms of the expectations of $\bar{h}_v^{l-1}$ and $\frac{\partial \bar{\mathcal{L}}}{\partial h_v^l}$ are upper-bounded: $||\mathbb{E}[\bar{h}_v^{l-1}]|| \leq M$, $||\mathbb{E}[\frac{\partial \bar{\mathcal{L}}}{\partial h_v^l}]|| \leq N$.*

Theorem 3. *Given a distributed full-graph (V, E) and optional bit-width set B, for each layer $l \in \{1, \cdots, L\}$ in the GNN, gradient variance upper bound Q^l in layer l is:*

$$Q^l = \sum_v^{|V|}\Big(\sum_{k_1}^{N_R(v)} \sum_{k_2}^{N_R(v)} \alpha_{k_1,v}^2 \alpha_{k_2,v}^2 \frac{D_{k_1}^{l-1} D_{k_2}^l (S_{k_{1_b}}^{l-1} S_{k_{2_b}}^l)^2}{6} + M^2 \sum_k^{N_R(v)} \alpha_{k,v}^2 \frac{D_k^l (S_{k_b}^l)^2}{6} + N^2 \sum_k^{N_R(v)} \alpha_{k,v}^2 \frac{D_k^{l-1} (S_{k_b}^{l-1})^2}{6}\Big) \tag{9}$$

where the definitions of $S_{k_b}^l$ and D_k^l can be found in Sec. 3.2. From a high-level view, for any v in the distributed full-graph (V, E), its neighborhood aggregation adds variance to model gradients in each layer if it has remote neighbors. For layer l, Q^l is decided by many factors: (i) **graph topology and partition strategy**, which determine the size of v's remote neighborhood $N_R(v)$; (ii) **GNN aggregation function**, corresponding to the aggregation coefficient $\alpha_{k,v}$ for each node; (iii) **dimension size and numerical range of remote message vectors** (D_k^l, the numerator in $S_{k_b}^l$); (iv) **choices of quantization bit-width** b_v for each node. Given factors (i)-(iii) which are decided by the GNN training job, we can choose (iv) the quantization bit-width accordingly to minimize the terms in the bound, and thus reduce the gradient variance and let training converge closer to the solution of Vanilla (Chen et al., 2021).

4.2 Bi-objective Optimization for Adaptive Bit-width Assignment

There is a trade-off in quantization bit-width assignment: using a larger quantization bit-width (e.g., 8-bit) leads to lower gradient variance upper bound Q^l in each layer, but less communication volume reduction (as compared to using 4-bit or 2-bit). Our goal is to design an adaptive bit assignment scheme for different messages between each device pair, to strike a good balance between training convergence and efficiency.

From Fig. 2 we know that the size of transferred data varies significantly across device pairs. We should also alleviate communication stragglers in each communication round.

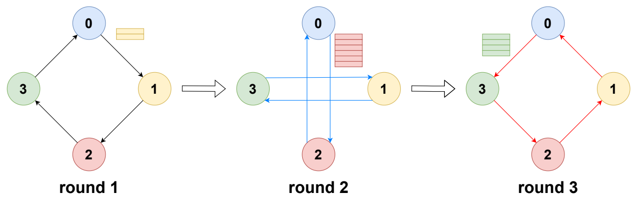


Figure 8. Illustration of ring all2all communication between four training devices. We only depict the data-sending process of device 0 for figure clarity.

Specifically, we follow (Wan et al., 2022b) to implement the all2all communication in a ring pattern (Fig. 8). For N devices, it takes $N-1$ communication rounds to finish the message exchange, where each of the devices sends/receives messages to/from its i-hop right/left neighbors in the ring during i-round communication. Let $B=\{2,4,8\}$ be the set of candidate bit-widths. For each communication round in the forward (backward) pass of GNN layer l, we formulate the following minimax optimization problem for bit-width selection:

$$\min_{b_k \in B} \max_{1\le i\le N} \theta_i \sum_{k}^{K_i} D_k^l b_k + \gamma_i \tag{10}$$

b_k denotes the bit-width assigned for quantizing message h_k^l. For any device pair i, K_i denotes the total number of messages to be transferred. D_l^k is the dimension of the remote message vector h_k^l. θ_i and γ_i are parameters of the cost model (Sarvotham et al., 2001). Problem 10 finds the device pair that has the longest predicted communication time and minimizes it.

According to Theorem 3, considering a message h_k^l in GNN layer l sent to a target device, we want to minimize the variance it brings to this layer's weight gradients. Apart from bit-width b_k, the dimension and the minimum (maximum) values of it (in $S_{k_b}^l$), and the sum of squares of all the aggregation coefficients $\sum_v^{N_T(k)} \alpha_{k,v}^2$ (where $N_T(k)$ denotes k's neighbors in the target device) allocated by its neighbors in the target device also influence the magnitude of variance. We solve the following minimization problem to minimize the total gradient variance in one communication round, where $\beta_k = \frac{\sum_v^{N_T(k)} \alpha_{k,v}^2 D_k^l (max(h_k^l)-min(h_k^l))^2}{6}$:

$$\min_{b_k \in B} \sum_i^N \sum_k^{K_i} \frac{\beta_k}{(2^{b_k}-1)^2} \tag{11}$$

We jointly address the two objectives in Eqn. 10 and Eqn. 11, which formulates a bi-objective optimization problem. We apply the weighted sum method to scalarize the objectives (Marler & Arora, 2004) and add auxiliary variables to convert it to a pure minimization problem:

$$\begin{aligned} &\min_{b_k \in B} \lambda \sum_i^N \sum_k^{K_i} \frac{\beta_k}{(2^{b_k}-1)^2} + (1-\lambda)Z,\ \lambda \in [0,1] \\ &s.t._{1\le i\le N}\ \ \theta_i \sum_k^{K_i} D_k^l b_k + \gamma_i \le Z,\ Z>0 \end{aligned} \tag{12}$$

Table 3. Graph datasets in our experiments.

Dataset	#Nodes	#Edges	#Features	#Classes	Size
Reddit	232,965	114,615,892	602	41	3.53GB
Yelp (Zeng et al., 2020)	716,847	6,977,410	300	100	2.10GB
ogbn-products	2,449,029	61,859,140	100	47	1.38GB
AmazonProducts	1,569,960	264,339,468	200	107	2.40GB

We convert the problem to a Mixed Integer Linear Program by viewing it as an Assignment Problem in the combinatorial optimization field and use off-the-shelf solvers (e.g., GUROBI (Gurobi Optimization, LLC, 2022)) to obtain the bit-width assignments. To better adapt to the value change of some parameters in β_k (e.g., the minimum and maximum values in message vectors) in training, we periodically re-solve the problem (Sec. 3.3). To reduce the overhead for solving the optimization, for one layer's forward or backward pass, we order messages transferred in each device pair according to their β_k values, and then divide them into groups to reduce the number of variables; messages in a group share the same bit-width assignment. We empirically set the size of message groups, which can be found in Appendix B.

5 Evaluation

Implementation. We build AdaQP on top of DGL 0.9 (Wang, 2019) and PyTorch 1.11 (Paszke et al., 2019), leveraging DGL for graph-related data storage and operations and using PyTorch distributed package for process group's initialization and communication. Before training begins, DGL's built-in METIS algorithm partitions the graph; each training process is wrapped to only one device (GPU) and broadcasts the remote node indices (built from DGL's partition book) to create sending and receiving node index sets for fetching messages from others. To support computation-communication parallelization, we integrate our customized *distributed graph aggregation layer* into PyTorch's Autograd mechanism. To support extremely low bit-width message compression, we follow (Liu et al., 2021b) to merge all the quantized messages with lower precision (4-bit or 2-bit) into uniform 8-bit byte streams. To transfer multiple bit-width quantized messages and match them with corresponding buffers, we first group messages according to their assigned bit-width, perform single bit-width quantization to each group and then concatenate all groups into a byte array for transmission. After communication, each training process recovers full-precision messages from the byte array based on a bit-retrieval index set maintained and updated by the Adaptive Bit-width Assigner.

Experimental Settings. We evaluate AdaQP on four large benchmark datasets (Hamilton et al., 2017a; Zeng et al., 2020; Hu et al., 2020), detailed in Table 3. The transductive graph learning task on Reddit and ogbn-products is single-label node classification, while the multi-label classification task is performed on Yelp and AmazonProducts. We use accuracy and F1-score (micro) as the model performance metric for these two tasks, respectively, and refer to them both as *accuracy*. All datasets follow the "fixed-partition" splits with METIS. We train two representative models, GCN (Kipf & Welling, 2017) and full-batch GraphSAGE (Hamilton et al., 2017a). To ensure a fair comparison, we unify all the model-related and training-related hyper-parameters throughout all experiments, whose details can be found in Appendix B. The model layer size and the hidden layers' dimensions are set to 3 and 256, respectively, and the learning rate is fixed at 0.01. Experiments are conducted on two servers (Ubuntu 20.04 LTS) connected by 100Gbps Ethernet, each having two Intel Xeon Gold 6230 2.1GHz CPUs, 256GB RAM and four NVIDIA Tesla V100 SXM2 32GB GPUs.

5.1 Expediting Training While Maintaining Accuracy

First, we show that AdaQP can drastically improve the training throughput while still obtaining high accuracy. We compare its performance with Vanilla and two other SOTA methods: PipeGCN (Wan et al., 2022b) and SANCUS (Peng et al., 2022), both of which show significant advantages over previous works (Jia et al., 2020; Cai et al., 2021; Tripathy et al., 2020; Thorpe et al., 2021). We implement Vanilla ourselves and use the open-source code of the other two to reproduce all the results. Note that PipeGCN only implements GraphSAGE while SANCUS implements GCN, so we only show their results on their respective supported GNNs. We run training of each model independently for three times, and the average and standard deviation are presented in Table 4. All the methods use the same number of training epochs.

We observe that AdaQP achieves the highest training speed and the best accuracy in the 14/16 and 12/16 sets of experiments, respectively. Specifically, AdaQP is $2.19 \sim 3.01\times$ faster than Vanilla with at most 0.30% accuracy degradation. By carefully and adaptively assigning bit-widths for messages, AdaQP can even improve accuracy up to 0.19%. Compared to AdaQP, PipeGCN and SANCUS not only are slower in most settings but also introduce intolerable model accuracy loss. This is because AdaQP does not rely on stale messages that slow down training convergence. What is more, properly applied quantization can benefit training due to the regularization effect of quantization noise introduced to model weights (Courbariaux et al., 2015).

Note that PipeGCN achieves higher training throughput than AdaQP on Reddit, which is because Reddit is much denser than others. This nature helps the cross-iteration pipeline design of PipeGCN (hiding communication overheads in computation) but does not always hold for other graphs. AdaQP does not rely on this prior property of graphs and can obtain consistent acceleration on distributed full-graph training. We also notice that SANCUS's performance is even worse than Vanilla's under many settings. This is because it adopts sequential node broadcasts, which is less efficient than the ring all2all communication pattern adopted by Vanilla.

5.2 Preserving Convergence Rate

To verify the theoretical analysis in Sec. 4 that AdaQP is able to maintain the same training convergence rate as Vanilla ($O(T^{-1})$), we show the training curves of all methods on Reddit and ogbn-products in Fig. 9 (the complete comparison can be found in Appendix C). We observe that our training curves almost coincide with those of Vanilla, verifying the theoretical training convergence guarantee in Sec. 4.1. On the other hand, both PipeGCN and SANCUS lead to slower training convergence, which is also consistent with their theoretical analysis (meaning that more training epochs will be needed if they intend to achieve the same accuracy as vanilla full-graph training and AdaQP). To further illustrate the end-to-end training expedition gains of AdaQP, we show the wall-clock time (the total training time, and for AdaQP, wall-clock time contains both the bit-width assignment time and the actual training time) of training among all the methods on AmazonProducts in Table 5. The complete comparison can be found in Appendix C.

5.3 Striking Better Trade-off with Adaptive Message Quantization

We compare our adaptive message quantization scheme with the uniform bit-width sampling scheme, which samples a bit-width from $\{2, 4, 8\}$ for each message uniformly and randomly. From Table 6, we see that adaptive message quantization obtains higher accuracy with faster training speed in almost all settings. By solving the bi-objective problem, adaptive message quantization can control the overall gradient variance to a certain level while taking into account the training speed, and alleviate stragglers in all communication rounds. However, uniform bit-width sampling is not as robust. In some cases, it leads to apparent accuracy degradation, e.g., 75.03% vs. 75.32%. This is because simply performing uniform bit-width sampling can easily assign lower bit-widths (2 or 4) to messages with large β values, thus introducing significant variance (Sec. 4) to model gradients and hurting the accuracy.

5.4 Time Breakdown

To understand the exact training throughput improvement and the extra overheads brought by AdaQP, we break down per-epoch training time into three parts (communication, computation and quantization time) and the wall-clock time into two parts (assignment time and actual training time).

Table 4. Training performance comparison among AdaQP and other works. The best **accuracy** and **training throughput** in each set of experiments are marked in bold.

Dataset	Partitions	Model	Method	Accuracy(%)	Throughput (epoch/s)	Dataset	Partitions	Model	Method	Accuracy(%)	Throughput (epoch/s)
Reddit	2M-1D	GCN	Vanilla	**95.36±0.03**	0.99	Yelp	2M-1D	GCN	Vanilla	**44.24±0.19**	1.18
			PipeGCN	†	†				PipeGCN	†	†
			SANCUS	94.73±0.17	1.11 (1.12×)				SANCUS	20.75±2.44	0.80
			AdaQP	**95.36±0.02**	**2.17 (2.19×)**				AdaQP	43.96±0.15	**3.04 (2.58×)**
		GraphSAGE	Vanilla	96.50±0.03	0.94			GraphSAGE	Vanilla	64.65±0.08	1.11
			PipeGCN	**96.62±0.00**	**3.72 (3.96×)**				PipeGCN	63.88±0.06	2.63 (2.37×)
			SANCUS	†	†				SANCUS	†	†
			AdaQP	96.49±0.02	2.13 (2.27×)				AdaQP	**64.72±0.13**	**3.15 (2.83×)**
	2M-2D	GCN	Vanilla	95.35±0.04	1.13		2M-2D	GCN	Vanilla	**43.86±0.62**	1.57
			PipeGCN	†	†				PipeGCN	†	†
			SANCUS	94.90±0.02	1.48 (1.31×)				SANCUS	20.78±0.2.45	0.66
			AdaQP	**95.38±0.03**	**2.65 (2.35×)**				AdaQP	43.84±0.63	**3.64 (2.32×)**
		GraphSAGE	Vanilla	96.55±0.03	1.16			GraphSAGE	Vanilla	64.67±0.12	1.19
			PipeGCN	**96.67±0.01**	**3.13 (2.70×)**				PipeGCN	63.73±0.14	2.32 (1.95×)
			SANCUS	†	†				SANCUS	†	†
			AdaQP	96.53 ± 0.04	2.65 (2.28×)				AdaQP	**64.78±0.05**	**3.58 (3.01×)**
ogbn-products	2M-2D	GCN	Vanilla	75.14±0.41	0.61	AmazonProducts	2M-2D	GCN	Vanilla	51.45±0.12	0.42
			PipeGCN	†	†				PipeGCN †	†	
			SANCUS	71.52±0.13	0.26				SANCUS	20.83±0.18	0.32
			AdaQP	**75.32±0.28**	**1.65 (2.70×)**				AdaQP	**51.50±0.08**	**1.16 (2.76×)**
		GraphSAGE	Vanilla	**78.90±0.17**	0.63			GraphSAGE	Vanilla	**75.69±1.32**	0.46
			PipeGCN	77.82±0.01	1.10 (1.75×)				PipeGCN	71.96±0.00	0.99 (2.15×)
			SANCUS	†	†				SANCUS	†	†
			AdaQP	78.85±0.20	**1.67 (2.65×)**				AdaQP	**75.69 ± 1.33**	**1.21 (2.63×)**
	2M-4D	GCN	Vanilla	75.11±0.09	0.79		2M-4D	GCN	Vanilla	51.38±0.16	0.58
			PipeGCN	†	†				PipeGCN	†	†
			SANCUS	71.99±0.16	0.21				SANCUS	21.22±0.07	0.27
			AdaQP	**75.30±0.17**	**2.18 (2.76×)**				AdaQP	**51.56±0.20**	**1.60 (2.76×)**
		GraphSAGE	Vanilla	78.89±0.09	0.77			GraphSAGE	Vanilla	75.80±1.16	0.62
			PipeGCN	76.67±0.01	1.10 (1.43×)				PipeGCN	71.91±0.00	1.02 (1.65×)
			SANCUS	†	†				SANCUS	†	†
			AdaQP	**78.90±0.08**	**2.15 (2.79×)**				AdaQP	**75.98±1.18**	**1.61 (2.60×)**

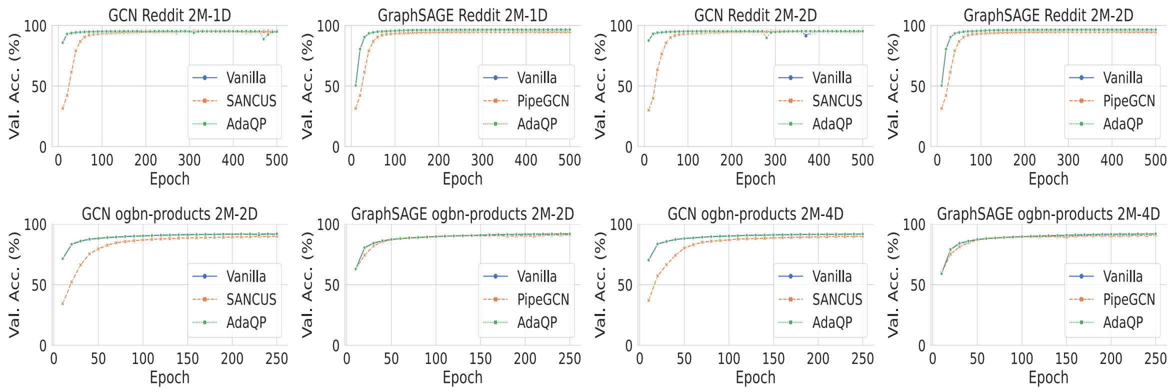


Figure 9. Epoch to validation accuracy comparison among vanilla full-graph training, PipeGCN, SANCUS and AdaQP.

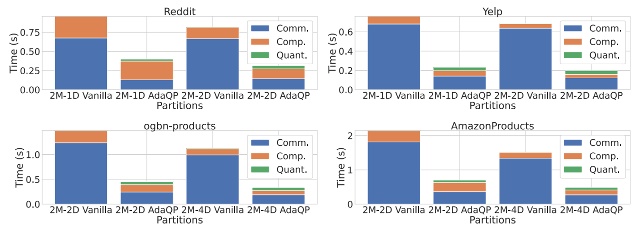
Table 5. Training wall-clock time comparison between AdaQP and other methods on AmazonProducts. The best **wall-clock time** is marked in bold.

Dataset	Partitions	Model	Method	Wall-clock Time (s)
AmazonProducts	2M-2D	GCN	Vanilla	2874.77
			PipeGCN	†
			SANCUS	3782.44
			AdaQP	**1053.51**
		GraphSAGE	Vanilla	2597.21
			PipeGCN	1212.65
			SANCUS	†
			AdaQP	**1008.34**
	2M-4D	GCN	Vanilla	2057.70
			PipeGCN	†
			SANCUS	3880.68
			AdaQP	**806.29**
		GraphSAGE	Vanilla	1927.85
			PipeGCN	1171.38
			SANCUS	†
			AdaQP	**771.52**

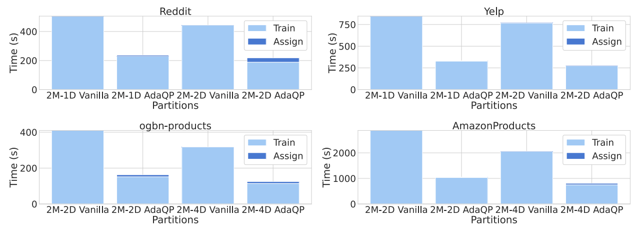
We provide the results of training GCN on all datasets in Fig. 10. For AdaQP, computation time only includes the marginal graph's computation time since that of the central graph is hidden within communication time (Sec. 2.2). Fig. 10(a) shows that compared to communication and computation time reduction benefits brought by AdaQP, the extra quantization cost is almost negligible. Specifically, for AdaQP, the overall quantization overheads are only 5.53% $\sim$ 13.88% of per-epoch training time, while the reductions in communication time and computation time are 78.29%

Table 6. Accuracy comparison between uniform bit-width sampling and adaptive message quantization on ogbn-products. The **best accuracy** is marked in bold.

Partitions	Model	Method	Accuracy (%)	Throughput (epoch/s)
2M-2D	GCN	Uniform	75.03±0.36	1.70
		Adaptive	**75.32±0.28**	1.65
	GraphSAGE	Uniform	78.84±0.23	1.64
		Adaptive	**78.85±0.20**	1.67
2M-4D	GCN	Uniform	75.16±0.16	2.14
		Adaptive	**75.30±0.17**	2.18
	GraphSAGE	Uniform	78.85±0.08	2.07
		Adaptive	**78.90±0.08**	2.15



(a) Per-epoch time



(b) Wall-clock time

Figure 10. Time breakdown of Vanilla and AdaQP. *quant.* denotes the sum of quantization and de-quantization times.

$\sim$ 80.94% and 13.16% $\sim$ 39.11%, respectively. Similar observations can be made in Fig. 10(b), where the average time overhead for bit-width assignment is 5.43% of the wall-clock time.

5.5 Senstivity Analysis

There are three hyper-parameters that determine the performance and overhead of adaptive message quantization in AdaQP: a) *group size* of messages, which determines the number of variables in Problem 12; b) λ, which decides the relative weight between time objective and variance objective in the bi-objective minimization problem; c) bit-width re-assignment *period*, which influences the total assignment overhead and the amount of traced data. We perform sensitivity experiments on these hyper-parameters by training GCN on 2M-4D partitioned ogbn-products (since this setting shows the largest accuracy gap between Vanilla and AdaQP). As shown in Fig. 11, for group size, the highest accuracy is obtained when it adopts the smallest value (50), which also brings much larger assignment overheads. As for λ, setting it to 0 or 1 both degrades the original problem to a single-objective problem, the best model accuracy is not achieved in these cases. As mentioned in Sec. 5.1, quantization can serve as a form of regularization; just rendering the lowest quantization variance ($\lambda = 1$) or just pursuing the highest throughput regarding the variance ($\lambda = 0$) is not the best choice to fully utilize the regularization effect of quantization. For the re-assignment period, a moderate period length (50) leads to the best model accuracy. How to automatically decide the best values for these hyper-parameters warantees further investigation, e.g., using a reinforcement learning method or searching for the best hyper-parameter combinations.

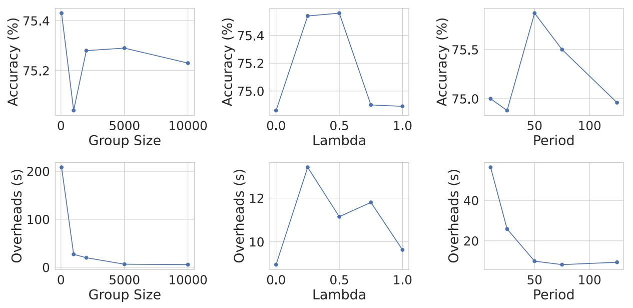


Figure 11. Sensitivity experiments on group size, λ, and re-assignment period.

Table 7. Training throughput on the 6M-4D partition.

Dataset	Method	Throughput (epoch/s)
ogbn-products	Vanilla	0.91
	AdaQP	**1.63 (1.79 $\times$)**
AmazonProducts	Vanilla	0.62
	AdaQP	**1.45 (2.34 $\times$)**

5.6 Scalability of AdaQP

We further evaluate AdaQP's training throughput on 6 machines connected by 100Gps Ethernet (two each have four NVIDIA Tesla V100 SXM2 32GB GPUs and four each have four NVIDIA Tesla A100 SXM4 40GB GPUs). We partition ogbn-products and AmazonProducts among the 24 devices and train GraphSAGE on them. Table 7 shows that AdaQP still achieves considerable throughput improvement in this 6M-4D setting, which validates its scalability.

6 Conclusion

We propose AdaQP, an efficient system for distributed full-graph GNN training. We are the first to reduce the substantial communication overhead with stochastic integer quantization. We further decompose the local graph partition residing on each device into a central graph and a marginal graph and perform computation-communication parallelization between the central graph's computation and the marginal graph's communication. We provide theoretical analysis to prove that AdaQP achieves similar training convergence rate as vanilla distributed full-graph training, and propose a periodically adaptive bit-width assignment scheme to strike a good trade-off between training convergence and efficiency with negligible extra overheads. Extensive experiments validate the advantages of AdaQP over Vanilla and SOTA works on distributed full-graph training.

Acknowledgements

This work was supported in part by grants from Hong Kong RGC under the contracts HKU 17208920, 17207621 and 17203522.

REFERENCES

Agarwal, S., Wang, H., Venkataraman, S., and Papailiopoulos, D. On the utility of gradient compression in distributed training systems. *Proceedings of Machine Learning and Systems*, 4:652–672, 2022.

Alistarh, D., Grubic, D., Li, J., Tomioka, R., and Vojnovic, M. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In *NIPS*, 2017.

Allen-Zhu, Z. Natasha: Faster non-convex stochastic optimization via strongly non-convex parameter. In *International Conference on Machine Learning*, pp. 89–97. PMLR, 2017.

Avriel, M. *Nonlinear programming: analysis and methods*. Courier Corporation, 2003.

Cai, Z., Yan, X., Wu, Y., Ma, K., Cheng, J., and Yu, F. Dgcl: an efficient communication library for distributed gnn training. *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021.

Chen, J., Gai, Y., Yao, Z., Mahoney, M. W., and Gonzalez, J. E. A statistical framework for low-bitwidth training of deep neural networks. *Advances in Neural Information Processing Systems*, 33:883–894, 2020.

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 *International Conference on Machine Learning*, pp. 1803–1813. PMLR, 2021.

Chen, J. J., Ma, T., and Xiao, C. Fastgcn: Fast learning with graph convolutional networks via importance sampling. *ArXiv*, abs/1801.10247, 2018.

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. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019.

Cong, W., Forsati, R., Kandemir, M., and Mahdavi, M. Minimal variance sampling with provable guarantees for fast training of graph neural networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 1393–1403, 2020.

Courbariaux, M., Bengio, Y., and David, J.-P. Binaryconnect: Training deep neural networks with binary weights during propagations. *Advances in neural information processing systems*, 28, 2015.

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. *2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)*, pp. 1044–1052, 2020.

Fu, F., Hu, Y., He, Y., Jiang, J., Shao, Y., Zhang, C., and Cui, B. Don't waste your bits! squeeze activations and gradients for deep neural networks via tinyscript. In *International Conference on Machine Learning*, pp. 3304–3314. PMLR, 2020.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. Neural message passing for quantum chemistry. *ArXiv*, abs/1704.01212, 2017.

Goodfellow, I., Bengio, Y., and Courville, A. *Deep learning*. MIT press, 2016.

Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2022. URL https://www.gurobi.com.

Hamilton, W. L., Ying, R., and Leskovec, J. Inductive representation learning on large graphs. In *NIPS*, 2017a.

Hamilton, W. L., Ying, Z., and Leskovec, J. Inductive representation learning on large graphs. In *NIPS*, 2017b.

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. *Advances in neural information processing systems*, 33:22118–22133, 2020.

Jia, Z., Lin, S., Gao, M., Zaharia, M. A., and Aiken, A. Improving the accuracy, scalability, and performance of graph neural networks with roc. In *MLSys*, 2020.

Kaler, T., Stathas, N., Ouyang, A., Iliopoulos, A.-S., Schardl, T., Leiserson, C. E., and Chen, J. Accelerating training and inference of graph neural networks with fast sampling and pipelining. *Proceedings of Machine Learning and Systems*, 4:172–189, 2022.

Karypis, G. and Kumar, V. Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. 1997.

Kipf, T. and Welling, M. Semi-supervised classification with graph convolutional networks. *ArXiv*, abs/1609.02907, 2017.

Liu, T., Chen, Y., Li, D., Wu, C., Zhu, Y., He, J., Peng, Y., Chen, H., Chen, H., and Guo, C. Bgl: Gpu-efficient gnn training by optimizing graph data i/o and preprocessing. *arXiv preprint arXiv:2112.08541*, 2021a.

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*, 2021b.

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 *USENIX Annual Technical Conference*, 2019.

Marler, R. T. and Arora, J. S. Survey of multi-objective optimization methods for engineering. *Structural and multidisciplinary optimization*, 26(6):369–395, 2004.

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. *Advances in neural information processing systems*, 32, 2019.

Peng, J., Chen, Z., Shao, Y., Shen, Y., Chen, L., and 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.

Robbins, H. and Monro, S. A stochastic approximation method. *The annals of mathematical statistics*, pp. 400–407, 1951.

Sarvotham, S., Riedi, R., and Baraniuk, R. Connection-level analysis and modeling of network traffic. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pp. 99–103, 2001.

Tailor, S. A., Fernández-Marqués, J., and Lane, N. D. Degree-quant: Quantization-aware training for graph neural networks. *ArXiv*, 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. *ArXiv*, abs/2105.11118, 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*, pp. 1–14. IEEE, 2020.

Wan, C., Li, Y., Li, A., Kim, N. S., and Lin, Y. Bns-gcn: Efficient full-graph training of graph convolutional networks with partition-parallelism and random boundary node sampling sampling. *Proceedings of Machine Learning and Systems*, 4, 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. *arXiv preprint arXiv:2203.10428*, 2022b.

Wang, M. Y. Deep graph library: Towards efficient and scalable deep learning on graphs. In *ICLR workshop on representation learning on graphs and manifolds*, 2019.

Wu, J., Huang, W., Huang, J., and Zhang, T. Error compensated quantized sgd and its applications to large-scale distributed optimization. In *ICML*, 2018.

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., and Philip, S. Y. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 32(1):4–24, 2020.

Xu, K., Hu, W., Leskovec, J., and Jegelka, S. How powerful are graph neural networks? *ArXiv*, abs/1810.00826, 2019.

Yu, Y., Wu, J., and Huang, L. Double quantization for communication-efficient distributed optimization. In *NeurIPS*, 2019.

Zeng, H., Zhou, H., Srivastava, A., Kannan, R., and Prasanna, V. K. Graphsaint: Graph sampling based inductive learning method. *ArXiv*, abs/1907.04931, 2020.

Zhang, M. and Chen, Y. Link prediction based on graph neural networks. In *NeurIPS*, 2018.

Zhang, Z., Cui, P., and Zhu, W. Deep learning on graphs: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 2020.

Zhu, F., Gong, R., Yu, F., Liu, X., Wang, Y., Li, Z., Yang, X., and Yan, J. Towards unified int8 training for convolutional neural network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 1969–1979, 2020.

A PROOF

A.1 Theorem 1

Proof. for any message vector h_v^l, the $round_{st}(\cdot)$ operation one of its elements $h_{v,i}^l$ is:

$$round_{st}(h_{v,i}^l) = \begin{cases} \lceil h_{v,i}^l \rceil & p = h_{v,i}^l - \lfloor h_{v,i}^l \rfloor \\ \lfloor h_{v,i}^l \rfloor & p = 1 - (h_{v,i}^l - \lfloor h_{v,i}^l \rfloor) \end{cases} \quad (13)$$

where p is the probability of rounding $h_{v,i}^l$ to certain value. $\lceil \cdot \rceil$ and $\lfloor \cdot \rfloor$ are ceil and floor operations respectively. Since $\lceil h_{v,i}^l \rceil - \lfloor h_{v,i}^l \rfloor = 1$, we have $\mathbb{E}[round_{st}(h_{v,i}^l)] =$

$\lceil h^l_{v,i}\rceil(h^l_{v,i}-\lfloor h^l_{v,i}\rfloor)+\lfloor h^l_{v,i}\rfloor(1-(h^l_{v,i}-\lfloor h^l_{v,i}\rfloor))=h^l_{v,i}$. Therefore, after $\tilde{q}_b(\cdot)$ (Eq.4) and $dq_b(\cdot)$ (Eq. 5) operations, the expectation of $\hat{h}^l_v$ is:

$$\begin{aligned}\mathbb{E}[\hat{h}^l_v] &= \mathbb{E}[round_{st}(\frac{h^l_v-Z^l_v}{S^l_{v_b}})S^l_{v_b}+Z^l_v]\\ &= S^l_{v_b}\mathbb{E}[round_{st}(\frac{h^l_v-Z^l_v}{S^l_{v_b}})]+Z^l_v\\ &= h^l_v\end{aligned} \tag{14}$$

as for variance of $\hat{h^l_v}$, we have:

$$\mathbb{V}ar[\hat{h}^l_v]=(S^l_{v_b})^2\mathbb{V}ar[round_{st}(\frac{h^l_v-Z^l_v}{S^l_{v_b}})] \tag{15}$$

let $h=\frac{h^l_v-Z^l_v}{S^l_{v_b}}$, since $\mathbb{V}ar[h]=\mathbb{E}[h^Th]-\mathbb{E}[h^T]\mathbb{E}[h]$, hence:

$$\begin{aligned}&\mathbb{V}ar[round_{st}(\frac{h^l_v-Z^l_v}{S^l_{v_b}})]=\sum_i^{D^l_v}\lceil h_i\rceil^2(h_i-\lfloor h_i\rfloor)\\ &+\lfloor h_i\rfloor^2(1-h_i+\lfloor h_i\rfloor)-h_i^2\\ &=\sum_i^{D^l_v}(2\lfloor h_i\rfloor h_i+h_i-\lfloor h_i\rfloor^2-\lfloor h_i\rfloor-h_i^2)\end{aligned} \tag{16}$$

we make the assumption that $\forall i, h_i-\lfloor h_i\rfloor=\sigma\sim$ Uniform(0,1), then $Var[round_{st}(\frac{h^l_v-Z^l_v}{S^l_{v_b}})]=\sum_i^{D^l_v}(\sigma-\sigma^2)=\frac{D^l_v}{6}$. Finally, the variance of $\hat{h}^l_v$ is:

$$\mathbb{V}ar[\hat{h}^l_v]=\frac{D^l_v(S^l_{v_b})^2}{6} \tag{17}$$

□

A.2 Theorem 2

Proof. We perform Taylor expansion for $\mathcal{L}(\mathbf{w}_{t+1})$ with Lagrangian Remainder:

$$\begin{aligned}f(\mathbf{w}_{t+1}) &= \mathcal{L}(\mathbf{w}_t-\alpha\mathbf{g}_t+\alpha\mathbf{g}_t-\tilde{\mathbf{g}}_t)\\ &= \mathcal{L}(\mathbf{w}_t-\alpha\mathbf{g}_t)+\alpha(\mathbf{g}_t-\tilde{\mathbf{g}}_t)^T\nabla\mathcal{L}(\mathbf{w}_t-\alpha\mathbf{g}_t)\\ &+\frac{1}{2}\alpha^2(\mathbf{g}_t-\tilde{\mathbf{g}}_t)^T\nabla^2\mathcal{L}(\epsilon_t)(\mathbf{g}_t-\tilde{\mathbf{g}}_t)\end{aligned} \tag{18}$$

since $\mathbb{E}[\tilde{\mathbf{g}}_t]=\mathbf{g}_t$ (from Assumption 1), we have:

$$\begin{aligned}&\mathbb{E}[\mathcal{L}(\mathbf{w}_{t+1})]\le\mathbb{E}[\mathcal{L}(\mathbf{w}_t-\alpha\mathbf{g}_t)\\ &+\alpha(\mathbf{g}_t-\tilde{\mathbf{g}}_t)^T\nabla\mathcal{L}(\mathbf{w}_t-\alpha\mathbf{g}_t)\\ &+\frac{1}{2}\alpha^2L_2||\mathbf{g}_t-\tilde{\mathbf{g}}_t||^2]\\ &\le\mathbb{E}[\mathcal{L}(\mathbf{w}_t-\alpha\mathbf{g}_t)]+\frac{1}{2}\alpha^2L_2Q^2\end{aligned} \tag{19}$$

where the first inequality is due to the property of Lipschitz continuity, the second inequality is due to the bounded variance property that $\mathbb{E}[||\mathbf{g}-\tilde{\mathbf{g}}||]\le Q$. We perform similar Taylor Expansion to $\mathcal{L}(\mathbf{w}_t-\alpha\mathbf{g}_t)$ in Eq. 19 and take expectation on both sides:

$$\begin{aligned}&\mathbb{E}[\mathcal{L}(\mathbf{w}_t-\alpha\mathbf{g}_t)]=\mathbb{E}[\mathcal{L}(\mathbf{w}_t)-\alpha\mathbf{g}_t^T\nabla\mathcal{L}(\mathbf{w}_t)\\ &+\frac{1}{2}\alpha^2\mathbf{g}_t^T\nabla\mathcal{L}(\mu_t)\mathbf{g}_t]\\ &\le\mathcal{L}(\mathbf{w}_t)-\alpha\mathbb{E}[||\mathbf{g}_t||^2]+\frac{1}{2}\alpha^2L_2\mathbb{E}[||\mathbf{g}_t||^2]\end{aligned} \tag{20}$$

denote $\mathbf{g}_t$ as $\nabla\mathcal{L}(\mathbf{w}_t)$ and plug Eq. 20 into Eq. 19, we have:

$$\begin{aligned}&(\alpha-\frac{1}{2}\alpha^2L_2)\mathbb{E}[||\nabla\mathcal{L}(\mathbf{w}_t)||^2]\le\mathbb{E}[\mathcal{L}(\mathbf{w}_t)]-\mathbb{E}[\mathcal{L}(\mathbf{w}_{t+1})]\\ &+\frac{1}{2}\alpha^2L_2Q^2\end{aligned} \tag{21}$$

since we assume that this problem exists global minimum (Assumption 1), Summing over t from 1 to T we have:

$$\frac{\sum_{t=1}^T\mathbb{E}[||\nabla\mathcal{L}(\mathbf{w}_t)||^2]}{T}\le\frac{2(\mathcal{L}(\mathbf{w_1})-\mathcal{L}^*)}{T(2\alpha-\alpha^2L_2)}+\frac{\alpha L_2Q^2}{2-\alpha L_2} \tag{22}$$

Viewing t as a random variable, we have Theorem 2.

□

A.3 Theorem 3

Proof. We denote model gradient matrix in GNNs' layer l with quantization variance as $\frac{\partial\mathcal{L}}{\partial\tilde{\mathbf{W}}^l}$, its full-precision counterpart as $\frac{\partial\mathcal{L}}{\partial\mathbf{W}^l}$, according to the forward pass form in Eq. 3, we have:

$$\begin{aligned}\frac{\partial\mathcal{L}}{\partial\tilde{\mathbf{W}}^l}&=\sum_v^{|V|}\sigma'(\cdot)\odot\frac{\partial\mathcal{L}}{\partial h^l_v}(\sum_u^{\{v\}\cup N(v)}\alpha_{u,v}h^{l-1}_u)^T\\ &=\sum_v^{|V|}\sigma'(\cdot)\odot(\sum_u^{\{v\}\cup N_L(v)}\alpha_{u,v}\frac{\partial\mathcal{L}}{\partial h^l_u}+\sum_k^{N_R(v)}\alpha_{k,v}\frac{\partial\mathcal{L}}{\partial\hat{h}^l_{k_b}})\cdot\\ &(\sum_u^{\{v\}\cup N_L(v)}\alpha_{u,v}h^{l-1}_u+\sum_k^{N_R(v)}\alpha_{k,v}\hat{h}^{l-1}_{k_b})^T\end{aligned} \tag{23}$$

Consider that each message quantization operation in the forward and backward pass are independent of each other, we can get the expectation and variance of $\frac{\partial\mathcal{L}}{\partial\tilde{\mathbf{W}}^l}$ based on Theorem 1:

$$\begin{aligned}\mathbb{E}[\frac{\partial \mathcal{L}}{\partial \tilde{\mathbf{W}}^l}] &= \sum_v^{|V|} \sigma'(\cdot) \odot (\sum_u^{\{v\}\cup N_L(v)} \alpha_{u,v} \frac{\partial \mathcal{L}}{\partial h_u^l} \\ &+ \sum_k^{N_R(v)} \alpha_{k,v} \mathbb{E}[\frac{\partial \mathcal{L}}{\partial \hat{h}_{k_b}^l}])(\sum_u^{\{v\}\cup N_L(v)} \alpha_{u,v} h_u^{l-1} \\ &+ \sum_k^{N_R(v)} \alpha_{k,v} \mathbb{E}[\hat{h}_{k_b}^{l-1}])^T \\ &= \frac{\partial \mathcal{L}}{\partial \mathbf{W}^l}\end{aligned} \tag{24}$$

for variance, we omit the activation function σ since it does not change variance form, we have:

$$\begin{aligned}\mathbb{V}ar[\frac{\partial \mathcal{L}}{\partial \tilde{\mathbf{W}}^l}] &= \sum_v^{|V|} \mathbb{V}ar[\frac{\partial \mathcal{L}}{\partial h_v^l}(\sum_u^{\{v\}\cup N(v)} \alpha_{u,v} h_u^{l-1})^T] \\ &= \sum_v^{|V|} \mathbb{E}[(\frac{\partial \mathcal{L}}{\partial h_v^l})^2]\mathbb{E}[(\sum_u^{\{v\}\cup N(v)} \alpha_{u,v} h_u^{l-1})^2]^T \\ &- \mathbb{E}[\frac{\partial \mathcal{L}}{\partial h_v^l}]^2 \mathbb{E}[(\sum_u^{\{v\}\cup N(v)} \alpha_{u,v} h_u^{l-1})^T]^2 \\ &= \sum_v^{|V|} Var[\frac{\partial \mathcal{L}}{\partial h_v^l}] Var[\sum_u^{\{v\}\cup N(v)} \alpha_{u,v} h_u^{l-1})] \\ &+ \mathbb{V}ar[\frac{\partial \mathcal{L}}{\partial h_v^l}]\mathbb{E}[\sum_u^{\{v\}\cup N(v)} \alpha_{u,v} h_u^{l-1})]^2 \\ &+ \mathbb{V}ar[\sum_u^{\{v\}\cup N(v)} \alpha_{u,v} h_u^{l-1})]\mathbb{E}[\frac{\partial \mathcal{L}}{\partial h_v^l}]^2\end{aligned} \tag{25}$$

Since the randomness is introduced by $N_R(v)$, utilizing Assumption 2 we have:

$$\begin{aligned}\mathbb{V}ar[\frac{\partial \mathcal{L}}{\partial \tilde{\mathbf{W}}^l}] &= \sum_v^{|V|} \mathbb{V}ar[\sum_k^{N_R(v)} \alpha_{k,v} \frac{\partial \mathcal{L}}{\partial \hat{h}_{k_b}^l}] \\ &\mathbb{V}ar[\sum_k^{N_R(v)} \alpha_{k,v} h_{k_b}^{l-1}] \\ &+ \mathbb{V}ar[\sum_k^{N_R(v)} \alpha_{k,v} \frac{\partial \mathcal{L}}{\partial \hat{h}_{k_b}^l}]\mathbb{E}[\sum_k^{\{v\}\cup N(v)} \alpha_{k,v} h_{k_b}^{l-1}]^2 \\ &+ \mathbb{V}ar[\sum_k^{N_R(v)} \alpha_{k,v} h_{k_b}^{l-1}]\mathbb{E}[\sum_k^{\{v\}\cup N(v)} \alpha_{k,v} \frac{\partial \mathcal{L}}{\partial \hat{h}_{k_b}^l}]^2 \\ &\leq \sum_v^{|V|} \mathbb{V}ar[\sum_k^{N_R(v)} \alpha_{k,v} \frac{\partial \mathcal{L}}{\partial \hat{h}_{k_b}^l}]\mathbb{V}ar[\sum_k^{N_R(v)} \alpha_{k,v} h_{k_b}^{l-1}] \\ &+ M^2 \mathbb{V}ar[\sum_k^{N_R(v)} \alpha_{k,v} \frac{\partial \mathcal{L}}{\partial \hat{h}_{k_b}^l}] + N^2 \mathbb{V}ar[\sum_k^{N_R(v)} \alpha_{k,v} h_{k_b}^{l-1}]\end{aligned} \tag{26}$$

use Theorem 1, we have:

$$\begin{aligned}\mathbb{V}ar[\frac{\partial \mathcal{L}}{\partial \tilde{\mathbf{W}}^l}] &\leq \sum_v^{|V|}(\sum_k^{N_R(v)} \alpha_{k,v}^2 \frac{D_k^l \cdot (S_{k_b}^l)^2}{6}) \\ &\cdot (\sum_k^{N_R(v)} \alpha_{k,v}^2 \frac{D_k^{l-1} \cdot (S_{k_b}^{l-1})^2}{6}) + M^2 \sum_k^{N_R(V)} \alpha_{k,v}^2 \frac{D_k^l (S_{k_b}^l)^2}{6} \\ &+ N^2 \sum_k^{N_R(v)} \alpha_{k,v}^2 \frac{D_k^{l-1} (S_{k_b}^{l-1})^2}{6})\end{aligned} \tag{27}$$

We can just let the model gradient matrix's upper bound Q^l in layer l be the gradient variance upper bound in Eqn. 27:

$$\begin{aligned}Q^l &= \sum_v^{|V|}(\sum_{k_1}^{N_R(v)} \sum_{k_2}^{N_R(v)} \alpha_{k_1,v}^2 \alpha_{k_2,v}^2 \frac{D_{k_1}^{l-1} D_{k_2}^l (S_{k_{1_b}}^{l-1} S_{k_{2_b}}^l)^2}{6} \\ &+ M^2 \sum_k^{N_R(v)} \alpha_{k,v}^2 \frac{D_k^l (S_{k_b}^l)^2}{6} + N^2 \sum_k^{N_R(v)} \alpha_{k,v}^2 \frac{D_k^{l-1} (S_{k_b}^{l-1})^2}{6})\end{aligned} \tag{28}$$

□

B TRAINING CONFIGURATION

We show the training configurations in Table 8, where GCN and GraphSAGE share the same configurations. We also include the message group size and the value of λ for AdaQP in the table.

C ADDITIONAL EXPERIMENTS

C.1 Training Convergence Comparison

Fig. 12 shows the training convergence comparison of all the methods on all the datasets under the same experiential settings of Sec. 5.1. As we can see, AdaQP consistently shows a fast convergence rate over other SOTA staleness-based expedition methods, which is similar to the conclusion drawn in Sec. 5.2.

C.2 Wall-clock Time Comparison

We provide the wall-clock time comparison of all the methods on all the datasets in Tab. 9 under the same experiment settings in Sec. 5.1. For the fairness of comparison, we include the extra bit-width assignment time overheads (details in 3.3) in the wall-clock time of AdaQP. The results prove that the extra overheads introduced by AdaQP are negligible compared to the overall wall-clock time reduction gains, which is consistent with the time breakdown analysis in Sec 5.4. We still achieve the shortest wall-clock time in 14/16 sets of experiments.

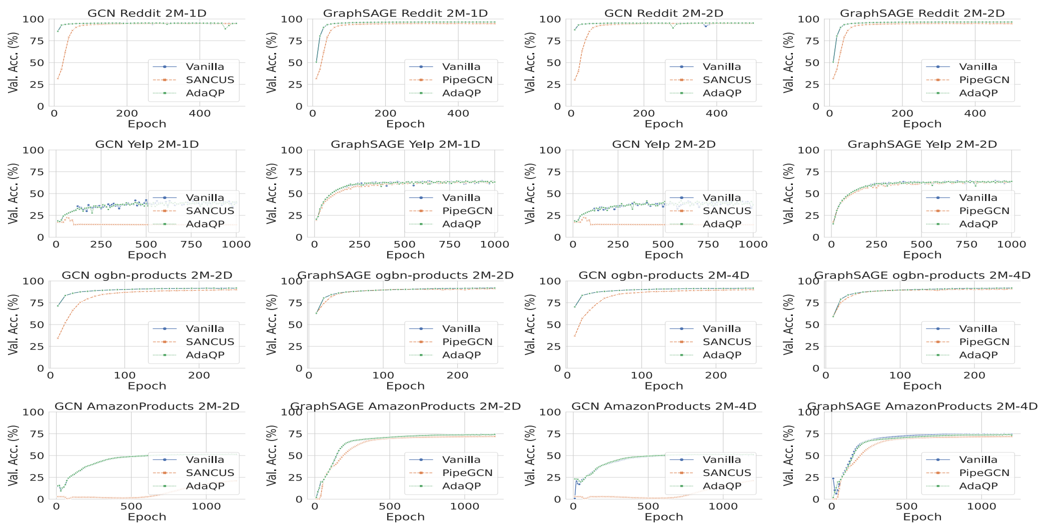


Figure 12. Epoch to validation accuracy comparison among Vanilla, PipeGCN, SANCUS, and AdaQP on all the datasets.

Table 8. Training configurations in our experiments.

Dataset	Model Layer	Hidden Dimension	Norm Function	Optimizer	Learning Rate	Dropout	Epoch	Message Group Size	λ
Reddit	3	256	LayerNorm	Adam	0.01	0.5	500	100	0.5
Yelp	3	256	LayerNorm	Adam	0.01	0.1	1000	1000	0.5
ogbn-products	3	256	LayerNorm	Adam	0.01	0.5	250	2000	0.5
AmazonProducts	3	256	LayerNorm	Adam	0.01	0.5	1200	500	0.5

Table 9. Training wall-clock time comparison between AdaQP and other methods, the best is denoted in bold.

Dataset	Partitions	Model	Method	Wall-clock Time (s)	Dataset	Partitions	Model	Method	Wall-clock Time (s)
Reddit	2M-1D	GCN	Vanilla	505.79	Yelp	2M-2D	GCN	Vanilla	846.79
			PipeGCN	†				PipeGCN	†
			SANCUS	447.28				SANCUS	1249.89
			AdaQP	**237.24**				AdaQP	**332.37**
		GraphSAGE	Vanilla	530.46			GraphSAGE	Vanilla	897.28
			PipeGCN	**135.29**				PipeGCN	381.11
			SANCUS	†				SANCUS	†
			AdaQP	246.71				AdaQP	**321.54**
	2M-2D	GCN	Vanilla	443.53		2M-4D	GCN	Vanilla	767.47
			PipeGCN	†				PipeGCN	†
			SANCUS	335.56				SANCUS	1509.41
			AdaQP	**218.14**				AdaQP	**281.77**
		GraphSAGE	Vanilla	429.85			GraphSAGE	Vanilla	839.96
			PipeGCN	**159.66**				PipeGCN	430.63
			SANCUS	†				SANCUS	†
			AdaQP	208.34				AdaQP	**289.23**
ogbn-products	2M-2D	GCN	Vanilla	409.54	AmazonProducts	2M-2D	GCN	Vanilla	2874.77
			PipeGCN	†				PipeGCN	†
			SANCUS	940.16				SANCUS	3782.44
			AdaQP	**162.53**				AdaQP	**1053.51**
		GraphSAGE	Vanilla	397.91			GraphSAGE	Vanilla	2597.21
			PipeGCN	229.11				PipeGCN	1212.65
			SANCUS	†				SANCUS	†
			AdaQP	**155.94**				AdaQP	**1008.34**
	2M-4D	GCN	Vanilla	317.48		2M-4D	GCN	Vanilla	2057.70
			PipeGCN	†				PipeGCN	†
			SANCUS	1186.68				SANCUS	3880.68
			AdaQP	**124.67**				AdaQP	**806.29**
		GraphSAGE	Vanilla	326.05			GraphSAGE	Vanilla	1927.85
			PipeGCN	229.31				PipeGCN	1171.38
			SANCUS	†				SANCUS	†
			AdaQP	**133.93**				AdaQP	**771.52**

A. Artifact Appendix

A.1 Abstract

Our artifact includes the full source code and training scripts of our paper. The Vanilla (the vanilla distributed full-graph training) is integrated into the AdaQP system. We provide both single-node and multi-node training examples. Running the code under the single-node multi-GPU settings requires a machine (256 GB RAM) equipped with Nvidia GPUs (32 GB each). To reproduce our experimental results, multiple machines with interconnect are needed. Software is provided with our dock image; one can also choose to install all dependencies with our setup bash script. Running provided scripts can validate core experiments in the paper, e.g., the throughput and accuracy, the training epoch time breakdown, and the bit-width assignment overhead analysis.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Graph Neural Networks, Bi-objective Optimization, Distributed training, Stochastic Integer Quantization
- **Data set:** Reddit, ogbn-products, Yelp, AmazonProducts
- **Run-time environment:** `IP`, `PORT`, `RANK` (set for torchrun), `GLOO_SOCKET_IFNAME` (set for multi-node training)
- **Hardware:** X86-CPU machines, Nvidia GPUs
- **Execution:** bash scripts, for both graph partitioning and training
- **Metrics:** throughput, accuracy, wall-clock time, time breakdown
- **Output:** stdout, log file
- **How much disk space required (approximately)?:** 50GB
- **How much time is needed to prepare workflow (approximately)?:** 40 minutes
- **How much time is needed to complete experiments (approximately)?:** 8 hours
- **Publicly available?:** yes
- **Code licenses (if publicly available)?:** MIT License
- **Workflow framework used?:** Pytorch, DGL
- **Archived (provide DOI)?:** 10.5281/zenodo.7783787

A.3 Description

A.3.1 How delivered

- Archival repository: https://doi.org/10.5281/zenodo.7783787
- GitHub repository: https://github.com/raywan-110/AdaQP
- docker image: https://hub.docker.com/r/raydarkwan/adaqp
- Approximate disk space: 50GB, used for datasets and graph partitions

A.3.2 Hardware dependencies

- several X86-CPU machines
- several Nvidia GPUs

A.3.3 Software dependencies

- Ubuntu 20.04 LTS
- Python 3.8
- CUDA 11.3
- Pytorch 1.11.0
- DGL 0.9.0
- OGB 1.3.3
- PuLP 2.6.0
- Gurobi 9.5.2
- Quant_cuda 0.0.0 (customized module)

A.3.4 Data sets

Reddit, ogbn-products, Yelp, AmazonProducts

A.4 Installation

Follow the instructions in `README.MD`. Using provided docker image is recommended. For example, by running the following commands:

```
docker pull raydarkwan/adaqp
docker run -it --gpus all --network=host
raydarkwan/adaqp
```

A.5 Experiment workflow

The workflow includes two steps: first, run graph partition scripts `scripts/partition/` to partition the graphs and store them into `data/part_data`; second, run scripts in `scripts/example` for single-node settings or run other scripts `scripts/*_all.sh` for multi-node settings to reproduce our experiment results. Detailed instructions are provided in `README.MD`.

A.6 Evaluation and expected result

All experimental results will be stored in `exp/` with different file formats (e.g., `*.csv`, `*.txt`, etc.).

A.7 Experiment customization

Adjust configurations in `AdaQP/config/*yaml` to customize dataset, model, training hyperparameter, bit-width assignment settings or add new configurations; adjust runtime arguments in `scripts/*` to customize graph partitions numbers, optional GPUs or machines for training, bit-width assignment strategies and training methods (AdaQP and its variants). Refer to `README.MD` for further details.

A.8 Notes

In our paper, we use two nodes, each equipped with 4 Nvidia NVIDIA Tesla V100 SXM2 32GB GPUs. To enable multi-node distributed training with torchrun, it may require setting the environment variable `GLOO_SOCKET_IFNAME` to the corresponding interfaces in the machine. For example, `export GLOO_SOCKET_IFNAME=eth0`.