Accelerating Distributed Training With Collaborative In-Network Aggregation

Jin Fang, Hongli Xu, *Member, IEEE*, Gongming Zhao, *Member, IEEE*, Zhuolong Yu, Bingchen Shen, and Liguang Xie, *Senior Member, IEEE*

***Abstract*— The surging scale of distributed training (DT) incurs significant communication overhead in datacenters, while a promising solution is in-network aggregation (INA). It leverages programmable switches (*e.g.*, Intel Tofino switches) for gradient aggregation to accelerate DT tasks. Due to switches' limited on-chip memory size, existing solutions try to design the memory sharing mechanism for INA. This mechanism requires gradients to arrive at switches synchronously, while network dynamics make it common for the asynchronous arrival of gradients, resulting in existing solutions being inefficient (*e.g.*, massive communication overhead). To address this issue, we propose GOAT, the first-of-its-kind work on gradient scheduling with collaborative in-network aggregation, so that switches can efficiently aggregate asynchronously arriving gradients. Specifically, GOAT first partitions the model into a set of sub-models, then decides which sub-model gradients each switch is responsible for aggregating exclusively and to which switch each worker should send its sub-model gradients. To this end, we design an efficient knapsack-based randomized rounding algorithm and formally analyze the approximation performance. We implement GOAT and evaluate its performance on a testbed consisting of 3 Intel Tofino switches and 9 servers. Experimental results show that GOAT can speed up the DT by $1.5\times$ compared to the state-of-the-art solutions.**

***Index Terms*— In-network aggregation, gradient scheduling, distributed training, datacenter network, programmable network.**

I. Introduction

With the increasing complexity of machine learning (ML) applications, such as computer vision [2], natural language processing [3] and recommender systems [4], the scale of ML tasks is growing explosively. In practice, distributed training (DT) [5], consisting of multiple workers and parameter servers (PS), is proposed to meet the needs for training large-scale ML tasks. In DT, workers train deep neural network (DNN) models locally and send gradients to the PS(s) for aggregation. After that, the PS(s) will send the aggregated gradients to workers. Due to a large volume of exchanged traffic during distributed training, communication overhead has become the main bottleneck [6], [7], [8], [9]. For example, for a DT task training BERT on 10Gbps links, 67% of the training time is occupied for communication [7].

Triggered by the recent rise of programmable networking [10], in-network aggregation (INA) [6], [7], [11], [12] has been proposed as a promising solution to alleviate the communication bottleneck. Instead of implementing aggregation purely in the PS(s), INA utilizes programmable switches (*e.g.*, P4-based [13] and FPGA-based [14]) to aggregate gradients within the network. Specifically, workers send gradients over the network, where programmable switches can aggregate gradients from multiple workers and send only the aggregated result to the PS. By doing so, INA helps to reduce the communication overhead from workers to the PS(s), increasing training throughput and speeding up distributed training [11].

The major challenge of INA is that programmable switches only have limited on-chip memory. A typical switch has tens of MBs memory size, while the gradient size of DNN models could be hundreds to thousands of MBs [7]. One intuitive solution is to increase the on-chip memory size directly. But it requires chip modification and raises the cost significantly. Alternatively, TEA [15] proposes the idea of extending the switch memory with external server memory. However, it does not consider the characteristics of INA workload (*e.g.*, the massive throughput demands) and will introduce a new bottleneck on the bandwidth towards the external memory.

To overcome the limitation of switch memory, existing INA solutions [6], [7], [16] design the complex *memory sharing mechanism* to enable gradient aggregation with programmable switches. However, *this kind of mechanism could be easily disturbed by synchronization delay and worker stragglers* [16]. For example, ATP [6] partitions the memory into isolated units, and each gradient fragment (a set of gradient elements) will be aggregated in a memory unit. Since the gradient size is larger than the size of programmable switch memory, one memory unit may be responsible for storing multiple gradient fragments. To guarantee the training correctness, programmable switches can not aggregate asynchronously

Manuscript received 25 November 2023; revised 6 March 2024; accepted 8 April 2024; approved by IEEE/ACM Transactions on Networking Editor R. Pedarsani. This work was supported in part by the National Science Foundation of China (NSFC) under Grant 62372426 and Grant 62102392, in part by the National Science Foundation of Jiangsu Province under Grant BK20210121, in part by Hefei Municipal Natural Science Foundation under Grant 2022013, in part by the Fundamental Research Funds for the Central Universities, and in part by the Youth Innovation Promotion Association of the Chinese Academy of Sciences under Grant 2023481. Some preliminary results of this paper were published in the Proceedings of IEEE/ACM IWQoS 2023 [DOI: 10.1109/IWQoS57198.2023.10188783]. *(Corresponding author: Hongli Xu.)*

Jin Fang, Hongli Xu, Gongming Zhao, and Bingchen Shen are with the School of Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui 230027, China, and also with Suzhou Institute for Advanced Research, University of Science and Technology of China, Suzhou, Jiangsu 215123, China (e-mail: fangjin98@mail.ustc.edu.cn; xuhongli@ustc.edu.cn; gmzhao@ustc.edu.cn; shenbcc@gmail.com).

Zhuolong Yu is with the Department of Computer Science, Johns Hopkins University, Baltimore, MD 21218 USA (e-mail: zhuolong@cs.jhu.edu).

Liguang Xie is with the Department of Computer Science, Virginia Tech, Blacksburg, VA 24061 USA (e-mail: xie@vt.edu).

Digital Object Identifier 10.1109/TNET.2024.3387948

arriving gradient fragments in the same memory unit, degrading the throughput of in-network aggregation (see Sec. III-A for details). In the extreme case, where all workers' gradients arrive at switches asynchronously, the PS will aggregate all gradients without performance gain of in-network aggregation., existing solutions may be inefficient for practical scenarios.

To deal with this issue, some works [6], [7] propose maintaining synchronization among workers. However, we argue that it will require considerable effort to keep the workers synchronized due to network dynamics. For instance, ATP [6] adopts ACK-based congestion control to modify the sending window of workers. However, it can not synchronize the pace of workers in time. So there still exists a considerable number of asynchronously arriving packets, leading to high aggregation overheads in the PS. Since network dynamics are common in datacenters [17], it is necessary to design alternative solutions to perform efficient in-network aggregation.

We find that the above solutions try to optimize memory utilization for each programmable switch individually, incurring inefficient aggregation in asynchronous scenarios. In practice, one model can be divided into a set of sub-models (*e.g.*, model layers), whose gradient will be aggregated independently [18]. Our key idea is to schedule sub-model gradients to multiple switches for collaborative in-network aggregation. This makes programmable switches possible to store all gradients and aggregate asynchronously arriving gradients.

To this end, this paper proposes GOAT, which *schedules sub-model gradients to multiple programmable switches for collaborative in-network aggregation*. Specifically, GOAT simultaneously decides (1) which sub-model gradients each programmable switch is responsible for aggregating exclusively and (2) to which programmable switches (or the PS) each worker should send these sub-model gradients. By doing these, GOAT can collaboratively utilize the on-chip memory of multiple switches and retain the efficiency of INA. However, it is non-trivial to realize GOAT. On the one hand, we utilize multiple programmable switches to aggregate gradients, thereby needing to balance the benefits of in-network aggregation and the routing costs of gradients to switches. On the other hand, the in-network aggregation will change the total amount of forwarded traffic, making existing routing methods [19] ineffective. Therefore, it is challenging to design an efficient gradient scheduling scheme with collaborative in-network aggregation to mitigate the communication bottleneck of DT. We summarize the main contributions of this paper as follows:

1) We design GOAT, the first-of-its-kind work that performs gradient scheduling with collaborative in-network aggregation to efficiently aggregate asynchronously arriving gradients and speed up the distributed training.
2) We formulate the problem of Gradient Scheduling for In-Network Aggregation (GINA) and present a knapsack-based randomized rounding algorithm, called KRGS, to solve this problem. KRGS achieves the approximation factor of $O(\log |S|)$, where $|S|$ is the number of programmable switches in the network. Under a proper assumption, *the bound of the approximation factor can be tightened to* 4.
3) We formulate the problem under the asynchronous distributed training scenario to determine the optimal number of participating workers in each epoch and prove its convergence. We present a greedy algorithm to solve the problem.
4) We conduct a small-scale testbed based on Intel Tofino switches and a large-scale simulation based on real-world network topologies. Both experimental and simulation results show that GOAT dramatically reduces the communication overhead by 81.2% and speeds up the distributed training by $1.5\times$ compared with the state-of-the-art solutions.

The rest of this paper is organized as follows. Section II presents some related works for this paper. Section III discusses the limitations of the related studies and explains our motivation. Section IV gives the formulation of Gradient Scheduling for In-Network Aggregation and proposes a knapsack-based randomized rounding algorithm. Section V formulates the problem under the asynchronous distributed training scenario and presents an efficient algorithm. Section VI illustrates the implementation of GOAT and Section VII evaluates the effectiveness of the proposed algorithm. We conclude the paper in Section VIII.

II. Related Work

This section first introduces the situation of distributed training. Then, we illustrate how to speed up the distributed training through in-network aggregation.

A. Distributed Training

A DNN model consists of multiple network layers, each of which contains a large number of parameters. Training a DNN model requires hundreds of epochs over the dataset to achieve convergence [20]. In each epoch, the compute node trains the DNN model over a dataset to generate the *gradient*, via a training algorithm (*e.g.*, stochastic gradient descent (SGD) or its variants [21], [22], [23], [24], [25]). We take SGD as an example. The compute node calculates gradient $g = \bigtriangledown F(w_t)$, where $\bigtriangledown$ denotes vector differential operator and $F(w_t)$ represents the value of loss function related to model w_t in epoch t, and updates the model parameter according to the gradient.

Distributed training [26] is proposed to split the whole dataset into multiple datasets, and parallelize the training with multiple compute nodes to speed up the DNN training.[1] For the data parallelism distributed training, there are two kinds of compute nodes: workers and parameter servers [5]. In each epoch, workers train the DNN model over their partition of the dataset to generate gradients and push gradients to parameter servers. Afterward, parameter servers aggregate all gradients with global aggregation algorithms [27], [28], [29], [30] and update the model parameters. For example, in synchronous SGD (SSGD) [31], the PS receives the gradients of workers and performs aggregation by calculating $\frac{1}{N}\sum_{n=1}^{N} g_t^n$, where

[1]There are two types of parallelism schemes: model parallelism and data parallelism. This paper focuses on the data parallelism distributed training.

N is the number of workers and g_t^n is the gradient of worker n in epoch t. At last, workers pull the updated results from parameter servers for the next training epoch.

As the scale of the distributed training task grows, the communication overhead becomes the main bottleneck [6], and in-network aggregation is proposed to reduce the traffic amount.

B. In-Network Aggregation

INA offloads part of gradients aggregated to programmable network devices (*e.g.*, Intel Tofino programmable switch [13]) to reduce the amount of transferred data, alleviating the communication bottleneck. A lot of works have focused on implementing in-network aggregation in programmable switches [6], [7], [11], [32]. For instance, SHARP [32] implements INA based on Mellanox's SiwtchIB-2 ASIC. iSwitch [11] designs an FPGA-based programmable switch to aggregate gradients of distributed reinforcement learning. As another option, P4-based programmable switches [13] attract a lot of attention. SwitchML [7] focuses on the in-network aggregation of a single rack and offloads gradient aggregation to the top-of-rack (ToR) switch. ATP [6] allows gradients to be aggregated either on near-worker ToR switches or near-PS ToR switches.

INA also exposes a novel system resource requirement: switch on-chip memory. The programmable switch needs to allocate its memory to store the intermediate results of gradient aggregation. Due to the gap between model size and on-chip memory size, current works [6], [7], [16], [33], [34] usually consider sharing memory for multiple model gradients (See Sec. III-A for details). For example, SwitchML [7] proposes to allocate the switch memory dedicated for a gradient aggregation until the switch finishes the job. INAlloc [33] proposes to periodically modify the switch memory allocation with the time granularity between a few seconds to a few minutes. ATP [6] ignores the newly arrived gradients if the corresponding memory area is occupied. ESA [16] designs a priority-based algorithm to decide which gradient will be allocated to the switch memory. However, these works mainly focus on how to allocate the on-chip memory of a single programmable switch. In fact, there are usually multiple programmable switches in datacenters. Considering that the gradients of model layers can be aggregated independently, the gradient aggregation can be scheduled into multiple programmable switches. Therefore, this paper designs GOAT to study the problem of gradient scheduling with collaborative in-network aggregation.

III. Motivation and GOAT Overview

This section first illustrates the pros and cons of the memory sharing scheme, then presents a motivating example. At last, we present the overview and workflow of GOAT.

A. Memory Sharing Scheme

Due to the limited size of switch memory, existing solutions [6], [7], [16] adopt the memory sharing scheme to conduct in-network aggregation. In particular, the switch memory is divided into N memory units, each of which can store a part of gradients (*i.e.*, gradient fragments) at a time, as illustrated in Fig. 1. Correspondingly, the gradient is divided into M fragments, each having the same size as the memory unit. When one fragment arrives at a switch, it will be hashed to a specific unit according to its index (*e.g.*, fragment i will be hashed to memory unit $i\%N$). Given that the switch memory size is usually smaller than the gradient size (*i.e.*, $N<M$), multiple gradient fragments will be hashed to the same memory unit, which means one memory unit needs to serve multiple gradient fragments during one DT job.

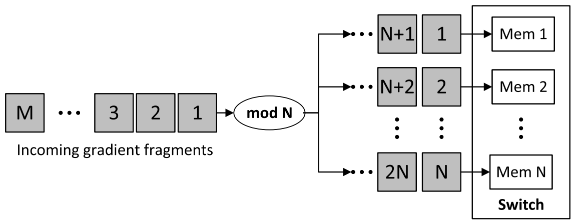


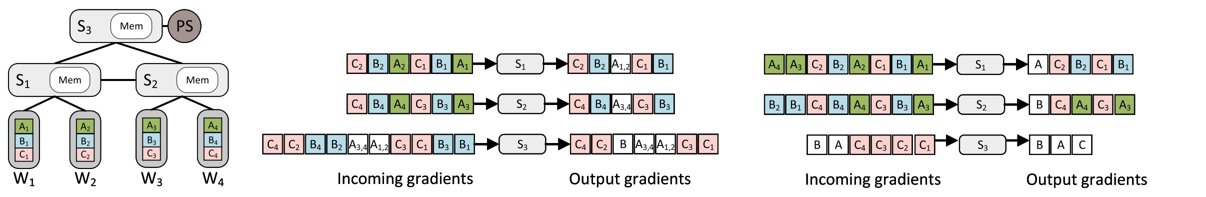
Fig. 1. Illustration of memory sharing scheme: the switch memory can be divided into N memory units. Correspondingly, a gradient can be partitioned into M gradient fragments, each assigned to a memory unit via hash function for aggregation. Since the size of switch memory is usually smaller than the size of gradient, multiple gradient fragments will be hashed to the same memory unit.

Ideally, workers' gradient fragments will arrive at the switch in sequence. For gradients hashed to the same memory unit, when one gradient fragment arrives, the memory unit has already released the previous gradient fragment, therefore no collision will happen. However, the memory sharing scheme may be inefficient when gradient fragments arrive at switches asynchronously. As mentioned above, each memory unit has room for only one fragment. Once a unit is occupied, it is unavailable to other fragments until it is released (*i.e.*, finish aggregating the current fragment). As a result, fragments encountered hash collision will be directly forwarded to the PS. In practice, it is common that gradients arrive at switches asynchronously because of network dynamics. Therefore, the memory sharing scheme will cause a significant volume of gradient fragments to be aggregated in the PS without fully utilizing the benefits of in-network aggregation.

B. A Motivating Example

Given that ATP is a popular INA solution with memory sharing, this section presents a motivating example to demonstrate the pros and cons of both ATP and GOAT. Consider a DT task with 1 PS, 4 workers (*i.e.*, W_1-W_4) and 3 programmable switches (*i.e.*, S_1-S_3), as shown in Fig. 2(a). For simplicity, we assume that worker W_i needs to send the gradient, divided into 3 fragments (*i.e.*, A_i, B_i and C_i) to the PS, and each switch's memory can store only one gradient fragment at a time.

We first introduce ATP, which performs the first-come-first-served strategy for gradient fragments allocated to the same memory unit [6]. In ATP, the near-worker switch needs to aggregate all gradients of connected workers and the near-PS switch needs to aggregate all gradients of downstream switches. For the synchronous scenario, the fragments of W_1 and W_2 arrive at S_1 with the sequence of $\{A_1, A_2, B_1, B_2, C_1, C_2\}$. S_1 first aggregates A_1 with the incoming fragment A_2 and outputs the aggregated fragment


S₃
Mem
PS
S₁
Mem
S₂
Mem
A₁ B₁ C₁
A₂ B₂ C₂
A₃ B₃ C₃
A₄ B₄ C₄
W₁ W₂ W₃ W₄
C₂ B₂ A₂ C₁ B₁ A₁ → S₁ → C₂ B₂ A₁,₂ C₁ B₁
C₄ B₄ A₄ C₃ B₃ A₃ → S₂ → C₄ B₄ A₃,₄ C₃ B₃
C₄ C₂ B₄ B₂ A₃,₄ A₁,₂ C₃ C₁ B₃ B₁ → S₃ → C₄ C₂ B A₃,₄ A₁,₂ C₃ C₁
A₄ A₃ C₂ B₂ A₂ C₁ B₁ A₁ → S₁ → A C₂ B₂ C₁ B₁
B₂ B₁ C₄ B₄ A₄ C₃ B₃ A₃ → S₂ → B C₄ A₄ C₃ A₃
B A C₄ C₃ C₂ C₁ → S₃ → B A C
Incoming gradients
Output gradients
Incoming gradients
Output gradients

(a) Network Topology (b) ATP (c) GOAT

Fig. 2. The left subplot shows the network topology of a distributed training task containing 1 PS, 4 workers (*i.e.*, W_1-W_4) and 3 programmable switches (*i.e.*, S_1-S_3). Worker W_i needs to send 3 gradient fragments (*i.e.*, A_i, B_i and C_i) to the PS. Each programmable switch can aggregate one gradient fragment at a time. In the middle and right subplots, we present the sequence of incoming and output fragments of programmable switches. We use $A_{1,2}$ to denote the aggregated fragment of A_1 and A_2. The middle subplot shows that S_3 outputs 7 gradient fragments to the PS and the right subplot shows that S_3 outputs 3 gradient fragments to the PS, which is optimal.

$A_{1,2}$. Then it aggregates B_1 with B_2 and outputs $B_{1,2}$. Finally it aggregates C_1 with C_2 and outputs $C_{1,2}$. S_2 and S_3 have similar processes.

However, due to network dynamics (*e.g.*, traffic congestion [35], [36] or equal-cost multipath routing [37], [38]), the traffic of W_1 and W_2 may arrive at S_1 asynchronously, where the incoming fragment sequence of S_1 is $\{A_1, B_1, C_1, A_2, B_2, C_2\}$, as shown in Fig. 2(b). In this scenario, S_1 first stores A_1, then directly forwards the subsequent fragments B_1 and C_1 to S_3 because these fragments do not match the stored fragment A_1. Finally, the output fragment sequence of S_1 is $\{B_1, C_1, A_{1,2}, B_2, C_2\}$. So does S_2. For S_3, it first receives and aggregates fragment B_1 with the incoming fragment B_3 and then buffers the intermediate aggregation result $B_{1,3}$ in its memory. Since S_3 cannot aggregates the subsequent fragments C_1 and C_3 with fragment $B_{1,3}$, it directly sends C_1 and C_3 to the PS. At last, the PS will receive 7 gradient fragments, *i.e.*, $\{C_1, C_3, A_{1,2}, A_{3,4}, B, C_2, C_4\}$. To handle asynchrony, workers regard the updated fragments from the PS as an ACK packet in ATP. Once a worker receives multiple out-of-order ACK packets, it regards that unreceived out-of-order packets have been lost. As a result, it will retransmit unreceived packets and modify the window size for synchronization. However, since this method needs extra RTTs for synchronizing worker sending rates [35], we argue that it cannot prevent massive traffic aggregated by the PS in time.

Since one programmable switch cannot store the entire gradient, we intend to utilize multiple programmable switches to aggregate gradients. Specifically, we schedule gradient fragments A, B and C to switches S_1, S_2 and S_3, respectively. In this way, workers W_1-W_4 need to send fragments A_1-A_4 to switch S_1, whose incoming fragment sequence is $\{A_1, A_2, B_1, B_2, C_1, C_2, A_3, A_4\}$, as shown in Fig. 2(c). S_1 will aggregate all gradient fragments A_i and forward the other fragments to the corresponding switches, *i.e.*, outputs $\{B_1, C_1, B_2, C_2, A\}$. For S_2, in addition to all fragments of W_3 and W_4, it also receives fragments B_1 and B_2 from S_1 and sends the aggregated fragment B along with the other fragments. The aggregation process of S_3 is similar to that of S_1 and S_2. As a result, the PS only receives 3 fragments, *i.e.*, $\{C, A, B\}$. This example shows that our scheme reduces the aggregation overhead of the PS by 57% (from 7 to 3) and the total communication overhead by 24% (from 17 to 13) compared with ATP. Thus, we conclude that scheduling gradients to perform collaborative in-network aggregation is more efficient than utilizing memory sharing mechanisms in asynchronous scenarios. Motivated by this example, we design the scheme of gradient scheduling with collaborative in-network aggregation, called GOAT.

Discussion. The above example illustrates the idea of gradient scheduling with collaborative in-network aggregation. In practice, one DNN model can be partitioned according to their model layers and spread sub-model to multiple devices to speed up the distributed training. For example, DADS [39] considers the DNN model layers as a directed graph and splits the DNN layers considering network conditions (*e.g.*, link bandwidth). DINA [40] proposes a fine-grained adaptive partitioning scheme which divides a DNN in pieces that can be smaller than a single layer to reduce the communication overhead. We can partition the model into sub-models with current methods [39], [40], [41], [42], [43], then schedule corresponding gradients to multiple switches for aggregation. Moreover, we intend to utilize a small number of programmable switches to aggregate a whole model collaboratively. For example, the gradient size of ResNet-50 [2] is 98MB and the memory size of Intel Tofino 2 [44] is 64MB. So it only takes 2 switches to aggregate ResNet-50's gradients. For large models, we can coexist with methods such as gradient quantization [45] to reduce the gradient size. Besides, considering that programmable switches are becoming popular in datacenters, we can aggregate large models with more switches if needed.

C. Overview of GOAT

Fig. 3 depicts the overview of GOAT, including the control plane and the data plane. Specifically, GOAT's control plane leverages the collected network information (*e.g.*, the available routing path set) and predefined model partition to determine the gradient scheduling policy, *i.e.*, to which programmable switches (or the PS) each worker should send its sub-model gradients. GOAT's data plane consists of workers, programmable switches and the PS. Workers divide models into a sub-model set and send gradients of sub-models to the PS. Programmable switches filter and aggregate the received gradient fragments. The PS is responsible for global aggregation.

Note that the core of GOAT is to determine the gradient scheduling policy, which will be described in Section IV. For the data plane, we can implement aggregation operations based on existing solutions [6], [7]. Due to space limitations, we omit the design details of the data plane and present the workflow in Sec. III-D.

D. Workflow of GOAT

Fig. 3 also describes the workflow of GOAT, which mainly consists of 5 steps as follows.

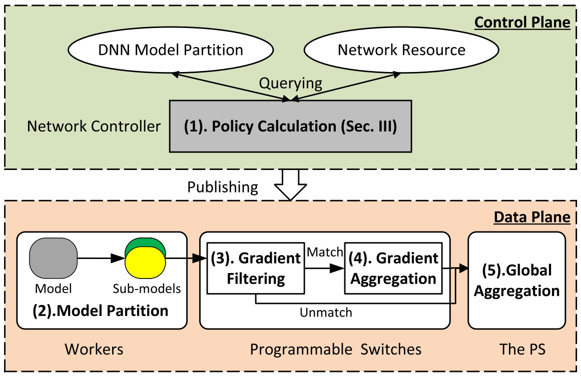


Fig. 3. System overview of GOAT. GOAT is composed of two parts. The control plane is responsible for determining the gradient scheduling policy. The data plane is responsible for collaborative in-network aggregation.

1) Policy calculation: The controller determines the gradient scheduling policy and publishes it to the data plane. Note that, the gradient scheduling policy is published only once, and the data plane will iteratively execute the following 4 steps in the DT task. In the following, we use *aggregation nodes* to represent programmable switches and the PS, since they will both aggregate gradients.
2) Model partition: Workers partition the model into sub-models and label the gradient fragments of sub-models according to the gradient scheduling policy. Each fragment will be identified by the tuple of <*aggregation node id, sub-model id, fragment id*>. The *aggregation node id* denotes the assigned aggregation node. The *sub-model id* represents the index of the sub-model. The *fragment id* represents the index of gradient fragments of the belonging sub-model.
3) Gradient filtering: Once a gradient fragment arrives, the programmable switch filters the fragment by matching the fragment's *aggregation node id* with its id. If they match, the programmable switch will perform gradient aggregation. Otherwise, the switch will directly forward the fragment according to the forwarding table.
4) Gradient aggregation: The programmable switches allocate the gradient fragments to specific memory units via HASH (<*aggregation node id, sub-model id, fragment id*>) $\bmod$ *memory size* for aggregation.
5) Global aggregation: The PS collects all gradient fragments (aggregated by programmable switches and directly sent from workers) and performs aggregation.

IV. Collaborative In-Network Aggregation

To achieve collaborative in-network aggregation, we first formulate the problem of Gradient Scheduling for In-Network Aggregation. Then we propose a knapsack-based randomized rounding algorithm named KRGS. At last, we analyze the approximation performance of KRGS.

A. System Model

Parameter Server Architecture. A parameter server architecture consists of the PS α and a worker set $W = \{w_1, w_2, \ldots, w_{|W|}\}$. Workers train models locally and send gradients to the PS for global aggregation.

TABLE I
Table of Notations

Notations	Semantics
α	the PS
W	the set of workers
G	the set of sub-model gradients
S	the set of programmable switches
T	the number of training epochs
$b(g)$	the size of the sub-model gradient g
$B(s)$	the on-chip memory size of programmable switch s
$D_w(s)$	the distance of worker w to aggregation node s
$D_s(\alpha)$	the distance of programmable switch s to the PS α
$H(t)$	the completion time of epoch t
$H(\alpha)$	the aggregation time of the PS α
$H(w,g,t)$	the arrival time of worker w's sub-model gradient g in epoch t
x_g^s	whether the gradient of the sub-model g is aggregated in aggregation node s or not
$y_{w,g}^s$	whether aggregation node s aggregates worker w's gradient g or not
z_w^t	whether the worker w participates the gradient aggregation in epoch t
K_t	the number of participating workers in epoch t
K_t^s	the number of participating workers aggregated by switch s in epoch t
z_w^t	whether the worker w participates the gradient aggregation in epoch t

DNN Model Training. A DNN model is partitioned into a set of sub-models, whose gradient can be denoted as $G = \{g_1, g_2, \ldots, g_{|G|}\}$. Each sub-model gradient has the size of $b(g)$, and is aggregated independently.

Programmable Network. We consider a datacenter containing four elements: a compute node set, a programmable switch set, a link set and a network controller.

1) Compute nodes host workers and the PS for model training and global aggregation.
2) Programmable switches are responsible for gradient filtering and aggregation. Let $S = \{s_1, s_2, \ldots, s_{|S|}\}$ denote the programmable switch set. Each switch s has a limited on-chip memory with the size of $B(s)$ to store gradients.
3) Compute nodes and programmable switches are connected via a set of links. We define the distance of two nodes as the number of links in the shortest path of two elements. Let $D_w(s)$ and $D_s(\alpha)$ denote the distance of worker w to aggregation node $s \in S \cup \{\alpha\}$ and the distance of programmable switch s to the PS α, respectively.
4) The network controller (*e.g.*, the PS) can be a logical controller used to manage the whole network, *e.g.*, deciding aggregation nodes of sub-model gradients.

B. Problem Formulation

This section describes the Gradient Scheduling for In-Network Aggregation (GINA) problem. The notations used in this paper are summarized in Table I. The key step of GINA is determining to which aggregation nodes each worker should send its sub-model gradients. Thus, let $y_{w,g}^s \in \{0, 1\}$ represent whether aggregation node s aggregates worker w's gradient g, or not. Let $x_g^s \in \{0,1\}$ represent whether sub-model gradient g is aggregated in aggregation node s, or not. The problem can be formulated as follows.

$$\min \sum_{g\in G}\Big(\sum_{w\in W}\sum_{s\in S\cup\{\alpha\}} y_{w,g}^s \cdot D_w(s) + \sum_{s\in S} x_g^s \cdot D_s(\alpha)\Big)\cdot b(g)$$
$$S.t. \begin{cases} \sum_{s\in S\cup\{\alpha\}} x_g^s \geq 1, & \forall g \in G \\ \sum_{s\in S\cup\{\alpha\}} y_{w,g}^s = 1, & \forall w \in W, g \in G \\ y_{w,g}^s \leq x_g^s, & \forall w \in W, g\in G, s \in S\cup\{\alpha\} \\ \sum_{g\in G} x_g^s \cdot b(g) \leq B(s), & \forall s \in S \\ x_g^s \in \{0,1\}, & \forall g\in G, s\in S\cup\{\alpha\} \\ y_{w,g}^s \in \{0,1\}, & \forall w \in W, g\in G, s\in S\cup\{\alpha\} \end{cases} \tag{1}$$

The first set of inequalities denotes that each gradient fragment should have a candidate aggregation node set. The aggregation nodes can be a programmable switch or the PS. The second set of equations represents that each worker needs to select one aggregation node for its sub-model gradients. The third set of inequalities guarantees that for each worker's sub-model gradient, the aggregation nodes are selected from the candidate aggregation node set. The fourth set of inequalities means that the size of stored gradient fragments in one programmable switch should be smaller than its memory size. Considering that communication is the main bottleneck of DT, our goal is to minimize the communication overhead in the network, including the non-aggregated gradients sent from workers to aggregation nodes and the aggregated gradients sent from programmable switches to the PS.

C. Algorithm Design

The path from workers to the PS can be splitted into two parts: the path from workers to programmable switches and the path from switches to the PS. Supposing that there is no in-network aggregation, the total traffic can be calculated as Eq. (2), by summing the traffic from workers to switches and the traffic from switches to the PS.

$$\sum_{w\in W}\sum_{g\in G}\sum_{s\in S\cup\{\alpha\}} y_{w,g}^s\cdot(D_w(s)+D_s(\alpha))\cdot b(g) \tag{2}$$

And our goal is converted to maximize the traffic reduced by in-network aggregation.

To minimize the total size of transferred gradients in the network, we need to maximize the traffic of aggregated gradients. Since the in-network aggregation mainly reduce the traffic among the second part of path (the switch receives multiple gradients but only needs to send one aggregated gradient to the PS), we subtract Eq. (2) from the objective of Eq. (1) to obtain the amount of traffic reduced by in-network aggregation.

$$\sum_{g\in G}\sum_{s\in S}\Big(\sum_{w\in W} y_{w,g}^s - x_g^s\Big)\cdot D_s(\alpha)\cdot b(g) + \sum_{g\in G}\sum_{w\in W} y_{w,g}^\alpha \cdot D_\alpha(\alpha)\cdot b(g) \tag{3}$$

Considering that the distance from the PS α to itself is zero (*i.e.*, $D_\alpha\alpha = 0$), we remove $\sum\limits_{g\in G}\sum\limits_{w\in W} y_{w,g}^\alpha \cdot D_\alpha(\alpha)\cdot b(g)$ in Eq. (3). As a result, we can convert Eq. (1) into maximizing the traffic amount of in-network aggregation as follows.

$$\max \sum_{g\in G}\sum_{s\in S}\Big(\sum_{w\in W} y_{w,g}^s - x_g^s\Big)\cdot D_s(\alpha)\cdot b(g)$$
$$S.t. \begin{cases} \sum_{s\in S\cup\{\alpha\}} x_g^s \geq 1, & \forall g \in G \\ \sum_{s\in S\cup\{\alpha\}} y_{w,g}^s = 1, & \forall w \in W, g \in G \\ y_{w,g}^s \leq x_g^s, & \forall w \in W, g\in G, s \in S\cup\{\alpha\} \\ \sum_{g\in G} x_g^s \cdot b(g) \leq B(s), & \forall s \in S \\ x_g^s \in \{0,1\}, & \forall g\in G, s\in S\cup\{\alpha\} \\ y_{w,g}^s \in \{0,1\}, & \forall w \in W, g\in G, s\in S\cup\{\alpha\} \end{cases} \tag{4}$$

We propose a knapsack-based randomized rounding algorithm to solve the converted GINA problem. Our algorithm consists of three steps. The first step relaxes Eq. (4) to a linear program by replacing $\{x_g^s, y_{w,g}^s\}$ with their fractional versions. We can solve it with a linear program solver (*e.g.*, PULP [46]) and the optimal solution is denoted as $\{\widetilde{x}_g^s, \widetilde{y}_{w,g}^s\}$. After that, we determine the set of assigned programmable switches for each sub-model gradient based on the optimal solution. For each gradient g, we first calculate $k(g) = \left\lfloor \sum_{s\in S} \widetilde{x}_g^s \right\rfloor$, which is the required number of programmable switches to aggregate gradient g. Then we put variables $\widetilde{x}_g^s$ $(\forall g \in G)$ into $k(g)$ knapsacks with min-max sum. For each knapsack a, sub-model gradient g will be scheduled to switch s with probability $\frac{\widetilde{x}_g^s}{\mathcal{S}_a}$, where $\mathcal{S}_a$ is the sum of x_g^s in knapsack a. We denote the set of assigned switches for sub-model gradient g as $S(g)$. Finally, for each worker w's gradient g, we calculate the probabilities of selecting switch $s \in S(g)$ to aggregate as $p_n(s) = \frac{\widetilde{y}_{w,g}^s}{\widetilde{x}_g^s}$ and of selecting the PS as $p_n(\alpha) = 1 - \sum_{s\in S(g)} p_n(s)$. Then we select an aggregation node $s \in S\cup\{\alpha\}$ with the probability of $p_n(s)$. The proposed algorithm is summarized in Alg. 1.

D. Performance Analysis

Theorem 1: Alg. 1 can guarantee that for each sub-model gradient, at least one aggregation node will be assigned.

Proof: We consider two situations according to whether the PS is selected as the optimal solution, or not. We first consider the situation that worker w's gradient g is not scheduled to the PS, *i.e.*, $\widetilde{x}_g^\alpha = 0$. According to the first set of inequalities in Eq. (4) and the definition of $k(g)$, we have:

$$1 \leq k(g) = \left\lfloor \sum_{s\in S\cup\{\alpha\}} \widetilde{x}_g^s \right\rfloor = \left\lfloor \sum_{s\in S} \widetilde{x}_g^s \right\rfloor \leq \sum_{s\in S} \widetilde{x}_g^s \tag{5}$$

Each gradient chooses one programmable switch from $k(g)$ knapsacks. Thus, there are $k(g)$ switches selected.

Algorithm 1 KRGS: Knapsack-Based Randomized Rounding for Gradient Scheduling

```
1:  Step 1: Solving the Relaxed Problem
2:  Construct a $LP$ by replacing with $x_g^s, y_{w,g}^s \in [0,1]$.
3:  Obtain the optimal solution $\{\widetilde{x}_g^s, \widetilde{y}_{w,g}^s\}$.
4:  Step 2: Assigning Switches for Sub-Model Gradients
5:  for each sub-model gradient $g \in G$ do
6:      Let $k(g) = \left\lfloor \sum_{s\in S} \widetilde{x}_g^s \right\rfloor$.
7:      Put $x_g^s$ $(\forall s \in S)$ into $k(g)$ knapsacks with min-max sum.
8:      for each knapsack $a$ do
9:          Let $\mathbb{A}$ denote the variables in knapsack $a$.
10:         Calculate $\mathcal{S}_a = \sum_{\widetilde{x}_g^s \in \mathbb{A}} \widetilde{x}_g^s$.
11:         Choose $s$ for $\widetilde{x}_g^s \in \mathbb{A}$ with probability $\frac{\widetilde{x}_g^s}{\mathcal{S}_a}$.
12:         Set $\widehat{x}_g^s = 1$ for chosen aggregation node $s$.
13:     end for
14:     Let $S(g) = \{s \in S | \widehat{x}_g^s = 1\}$ denote the set of switches
        responsible for aggregating sub-model gradient $g$.
15: end for
16: Step 3: Determining Aggregation Nodes for Workers' Sub-
    Model Gradients
17: for each worker $w \in W$ do
18:     for each gradient $g \in G$ do
19:         Set the probabilities of selecting switch $s \in S(g)$ and the
            PS to $p_n(s) = \frac{\widetilde{y}_{w,g}^s}{\widetilde{x}_g^s}$ and $p_n(\alpha) = 1 - \sum_{s\in S(g)} p_n(s)$,
            respectively.
20:         Select an aggregation node $s \in S \cup \{\alpha\}$ with the probability
            of $p_n(s)$.
21:     end for
22: end for
```

We then consider the situation that worker w's gradient g is scheduled to the PS, *i.e.*, $\widetilde{x}_g^\alpha = 1$. We have:

$$0 \le k(g) - 1 = \left\lfloor \sum_{s\in S} \widetilde{x}_g^s \right\rfloor \le \sum_{s\in S} \widetilde{x}_g^s \tag{6}$$

According to line 17 of Alg. 1, even if each gradient does not choose one switch for aggregation, it will be aggregated in the PS ($p_n^\alpha = 1 - \sum\limits_{s\in S(g)} p_n(s) = 1$).

As a result, we can guarantee that for each sub-model gradient, there is at least one node for aggregation. ■

Theorem 2: Alg. 1 guarantees that for each worker, each sub-model gradient is aggregated in one aggregation node.

Proof: According to line 17 of Alg. 1, for each worker's sub-model gradient, the sum of probabilities of selecting aggregation nodes is:

$$\sum_{s\in S(g)} p_n(s) + (1 - \sum_{s\in S(g)} p_n(s)) = 1, \forall w \in W \tag{7}$$

Eq. (7) shows that, each sub-model gradient will select one aggregation node. Therefore, the theorem holds. ■

Lemma 3: For each knapsack a, the lower bound of $\mathcal{S}_a$ is greater than 0.5.

Proof: By the definition of $k(g)$, we have:

$$\sum_{s\in S} \widetilde{x}_g^s = k(g) + \varepsilon, 0 < \varepsilon < 1 \tag{8}$$

Then, we define two sets as follows:

$$\begin{cases} \mathcal{X}_1 = \{\widetilde{x}_g^s | 0.5 < \widetilde{x}_g^s < 1, s \in S\} \\ \mathcal{X}_2 = \{\widetilde{x}_g^s | 0 < \widetilde{x}_g^s < 0.5, s \in S\} \end{cases} \tag{9}$$

Supposing that we select two variables denoted as x_m^1 and x_m^2 from $\mathcal{X}_2$ randomly. The value of $x_m^3 = x_m^1 + x_m^2$ is either greater than 0.5 or less than 0.5. If $x_m^3 > 0.5$, then $\mathcal{X}_1 = \mathcal{X}_1 + x_m^3$ and $\mathcal{X}_2 = \mathcal{X}_2 - \{x_m^1, x_m^2\}$. Otherwise, $\mathcal{X}_2 = \mathcal{X}_2 - \{x_m^1, x_m^2\} + x_m^3$. We repeat the above operations until $|\mathcal{X}_2| \le 1$. Supposing that there is one variable in $\mathcal{X}_2$, there are at most $k(g) - 1$ variables in $\mathcal{X}_1$. According to the definition of $\mathcal{X}_1$, the value of variables in $\mathcal{X}_1$ are all less than 1. Thus, we have:

$$\sum_{\widetilde{x}_g^s \in \mathcal{X}_1} \widetilde{x}_g^s < k(g) - 1 \tag{10}$$

$$\sum_{\widetilde{x}_g^s \in \mathcal{X}_2} \widetilde{x}_g^s < 0.5 \tag{11}$$

Combining Eq. (10) and Eq. (11), we have:

$$\sum_{s\in S} \widetilde{x}_g^s < k(g) - 0.5 \tag{12}$$

However, Eq. (12) contradicts Eq. (8). Thus, there are at least $k(g)$ variables in $\mathcal{X}_1$. Since we put variables to knapsacks with the min-max sum, the sum of knapsack a must be greater than 0.5. ■

In order to facilitate the description of approximation analysis, we give a famous lemma for probability analysis.

Lemma 4: **Chernoff Bound** [47]: Given n independent variables: $y_1, y_2, \ldots, y_n, \forall y_i \in [0,1]$. Let $\tau = \mathbb{E}\left[\sum_{i=1}^n y_i\right]$. Then, $\Pr\left[\sum_{i=1}^n y_i \ge (1+\varrho)\tau\right] \le e^{\frac{-\varrho^2 \tau}{2+\varrho}}$, where ϱ is an arbitrary positive value.

Theorem 5: Alg. 1 will not exceed the memory constraint of programmable switches by an approximation factor of $O(\log |S|)$.

Proof: We first prove that for each gradient $g \in G$ and programmable switch $s \in S$, we have $\mathbb{E}\left[\widehat{x}_g^s\right] \le 2 \cdot \widetilde{x}_g^s$. According to Alg. 1, we choose switch s with the probability of $\frac{\widetilde{x}_g^s}{\mathcal{S}_a}$, thereby $\mathbb{E}\left[\widehat{x}_g^s\right] = \frac{\widetilde{x}_g^s}{\mathcal{S}_a}$. According to Lemma 3, we can obtain $\mathbb{E}\left[\widehat{x}_g^s\right] = \frac{\widetilde{x}_g^s}{\mathcal{S}_a} \le 2 \cdot \widetilde{x}_g^s$.

Then we define $\delta_m^s = \widehat{x}_g^s \cdot b(g)$ as the size of gradient g aggregated in the programmable switch s. Since each gradient g selects the programmable switch s independently, we have $\mathbb{E}\left[\sum_{g\in G} \delta_s\right] = \sum_{g\in G} \frac{\widetilde{x}_g^s}{\mathcal{S}_a} \cdot b(g)$. By the definition of δ_s, we can get the expected workload of each programmable switch $s \in S$:

$$\begin{aligned} \mathbb{E}\left[\sum_{g\in G} \delta_s\right] &= \sum_{g\in G} \frac{\widetilde{x}_g^s}{\mathcal{S}_a} \cdot b(g) \\ &\le 2 \cdot \sum_{g\in G} \widetilde{x}_g^s \cdot b(g) \le 2 \cdot B(s) \end{aligned} \tag{13}$$

Let $B_{\min}$ denote the minimum memory capacity among the programmable switches. We then define a constant value $\nu = \min\{\frac{2 \cdot B_{\min}}{b(g)}, \forall g \in G\}$ to normalize the expected on-chip memory workload. Combining Eq. (13) and the definition of ν, we have:

$$\begin{cases} \frac{\delta_s \cdot \nu}{2 \cdot B(s)} \in [0,1] \\ \mathbb{E}\left[\sum_{g \in G} \frac{\delta_s \cdot \nu}{2 \cdot B(s)}\right] \leq \nu \end{cases} \tag{14}$$

By applying Lemma 4, we have:

$$\Pr\left[\sum_{g \in G} \frac{\delta_s \cdot \nu}{2 \cdot B_s} \geq (1+\varrho) \cdot \nu\right] \leq e^{\frac{-\varrho^2 \nu}{2+\varrho}}$$
$$\Rightarrow \Pr\left[\sum_{g \in G} \frac{\delta_s}{2 \cdot B_s} \geq (1+\varrho)\right] \leq e^{\frac{-\varrho^2 \nu}{2+\varrho}} \tag{15}$$

We want to find ϱ for which the probability upper bound above becomes very small. Specifically, we assume that:

$$\Pr\left[\sum_{g \in G} \frac{\delta_s}{2 \cdot B_s} \geq (1+\varrho)\right] \leq e^{\frac{-\varrho^2 \nu}{2+\varrho}} \leq \frac{1}{|S|} \tag{16}$$

which means that the upper bound approaches quickly to zero as the network grows. By solving Eq. (16), we have:

$$\varrho \geq \frac{\log|S| + \sqrt{\log^2|S| + 8\nu \log|S|}}{2\nu}, (|S| \geq 2)$$
$$\Rightarrow \varrho \geq \frac{\log|S|}{\nu} + 2, (|S| \geq 2) \tag{17}$$

In practice, the on-chip memory size of Intel Tofino 2 is 64MB [44]. According to the default model partition of BERT in PyTorch [48], the average size of sub-model gradients is 2MB, *i.e.*, $b(g) = 2$. Under this setting, $\nu = \frac{2 \cdot 64}{2} \approx 64$. We assume the number of programmable switches in a datacenter is $|S| = 20$ [49], so $3 \cdot \log|S| \approx 3.9$. Combining these assumptions, we can obtain that $\nu \geq 3 \cdot \log|S|$. As a result, we have:

$$\varrho \geq \frac{\log|S| + \sqrt{\log^2|S| + 8\nu \log|S|}}{2\nu}$$
$$\Rightarrow \varrho \geq \frac{\log|S| + \sqrt{(2\nu - \log|S|)^2 + 12\nu \log|S| - 4\nu^2}}{2\nu}$$
$$\Rightarrow \varrho \geq \frac{\log|S| + \sqrt{(2\nu - \log|S|)^2}}{2\nu} \Rightarrow \varrho \geq 1 \tag{18}$$

We can draw that the approximate factor of the memory constraint is $2 \cdot (\varrho + 1) = \frac{2 \cdot \log|S|}{\nu} + 6 = O(\log|S|)$. Under the proper assumption (*i.e.*, $\nu \geq 3 \cdot \log|S|$), the bound can be tightened to 4. ■

Theorem 6: After rounding, the communication overhead will not exceed the fractional solution by an approximate factor of $O(\log|W \cdot S|)$.

Proof: We first prove that for each worker w's gradient $g \in G$ and programmable switch $s \in S$, we have $\mathbb{E}\left[\widehat{y}^s_{w,g}\right] \leq 2 \cdot \widetilde{y}^s_{w,g}$. For each worker w, we choose the switch s to aggregate gradient g with the probability of $\frac{\widetilde{y}^s_{w,g}}{\widetilde{x}^s_g}$, where switch s must be assigned for aggregating gradient g. Thus, we have $\mathbb{E}\left[\widehat{y}^s_{w,g}\right] = \frac{\widetilde{y}^s_{w,g}}{\widetilde{x}^s_g} \cdot \frac{\widetilde{x}^s_g}{S_a}$. According to Lemma 3, we can obtain $\mathbb{E}\left[\widehat{y}^s_{w,g}\right] = \frac{\widetilde{y}^s_{w,g}}{S_a} \leq 2 \cdot \widetilde{y}^s_{w,g}$. Then we can analyze the approximation ratio performance based on the randomized rounding method. Since the proof process is similar to that of Theorem 5, we omit it here. ■

V. Aggregate With Asynchronous Training

In this section, we show that GOAT can be adopted into the asynchronous training scenario to speed up distributed training tasks. We first provide the problem formulation of collaborative in-network aggregation in asynchronous training scenario. Then, we analyze the convergence of collaborative in-network aggregation under asynchronous training. At last, we present an efficient algorithm to solve the problem.

A. Problem Formulation

Although we can aggregate asynchronously arrived gradients with collaborative in-network aggregation, the PS still needs to wait for the gradients of stragglers. In this section, to facilitate the distributed training, we allow the PS to aggregate the gradients of the fastest K_t workers in each epoch, where K_t represents the number of participating workers in epoch t. We use $z^t_w \in \{0,1\}$ to denote whether the worker w participates in the gradient aggregation in epoch t or not. Let $H(w,g,t)$ denote the arrival time of worker w's sub-model gradient g in epoch t. The completion time of epoch t is calculated as:

$$H(t) = \max\{z^t_w \cdot H(w,g,t)\} + H(\alpha), \forall t \in [T] \tag{19}$$

where $H(\alpha)$ denotes the global aggregation time of the PS.

Eq. (19) means that the duration of each epoch is regarded as the longest arrival time among participated workers plus the aggregation time. Since the global aggregation time takes up a small ratio during the training procedure [11], here we assume the aggregation time is stable among all training epochs (*i.e.*, $H(\alpha)$ is a constant). According to the above definitions, we formulate the problem as the following:

$$\min \sum_{t=1}^{T} H(t)$$
$$S.t. \begin{cases} F(x_T) \leq \mathcal{F} & \\ H(t) = \max\{z^t_w \cdot H(w,g,t)\} + H(\alpha), & \forall t \in [T] \\ K_t = \sum_{w \in W} z^t_w, & \forall t \in [T] \\ K^s_t = \sum_{w \in W} \sum_{g \in G} y^s_{w,g} \cdot z^t_w, & \forall t, s \\ z^t_w \in \{0,1\}, & \forall t, w \end{cases} \tag{20}$$

The first set of inequalities expresses the convergence requirement of workers, where $F(x_T)$ is a user-specified loss function (*e.g.*, linear regression [50], logistic regression [51], or support vector machine [52]), and $\mathcal{F}$ is the convergence threshold of the loss value after T training epochs. The second set of equations denotes the completion time of epoch t. The third and fourth sets of equations mean that the PS and switch

s aggregate K_t and K_t^s gradients in epoch t, respectively. Our goal is to minimize the completion time of the distributed training task.

B. Proof of Convergence

To analyze the feasibility of collaborative in-network aggregation with asynchronous training, we present the convergence analysis in this section.

Assumptions. In the datacenter, the training dataset is usually uniformly partitioned and distributed to workers independently [6], [7], [12], we assume that each worker trains the model on the independent and identical distributed dataset. The objective is to minimize the result of the loss function as follows:

$$\min_{x\in\mathbb{R}^d} F(x) = \frac{1}{n}\sum_{i=1}^{n} \mathbb{E}_{\xi_w\sim\mathbb{D}_w}\left[f(x,\xi_w)\right] \tag{21}$$

where $x \in \mathbb{R}^d$ is the model parameters shared by all workers, and $\mathbb{D}_w$ represents the local dataset of worker w, and $f(x,\xi_w)$ is the value of loss function associated with model parameters x and a batch ξ_w of the local dataset $\mathbb{D}_w$. Then, our analysis is based on three popular assumptions, which are widely used in literatures [53], [54], and [55].

Assumption 1 (Smoothness): The loss function f is smooth with a constant $L > 0$, which implies that $\forall x, y$, we have:

$$f(y) - F(x) \le \langle \nabla F(x), y - x\rangle + \frac{L}{2}\|y-x\|^2 \tag{22}$$

Assumption 2 (Unbiased Local Gradient): For each worker w with local data, the stochastic gradient is locally unbiased:

$$\mathbb{E}_{\xi_w\sim\mathbb{D}_w}\left[\nabla f(x,\xi_w)\right] = \nabla f_w(x) \tag{23}$$

Assumption 3 (Bounded Variance): The variance of the local gradient of worker w is bounded uniformly by:

$$\mathbb{E}_{\xi_w\sim\mathbb{D}_w}\left[\|\nabla F(x,\xi_w) - \nabla F_w(x)\|^2\right] \le \sigma^2 \tag{24}$$

and the variance between local and global gradient satisfies:

$$\|\nabla F_i(x) - \nabla F(x)\|^2 \le \beta^2 \tag{25}$$

where β^2 quantifies the deviation between local and global gradient. We consider $\beta^2 = 0$ when the training data are identically distributed among workers [55].

Convergence analysis. We first provide the following useful lemma [55] to characterize the gradient bias caused by ignoring the aggregation of straggler gradients.

Lemma 7: The variance between the average gradient of $|W|$ workers and K workers is characterized as follows:

$$\mathbb{E}\left[\|\bar{G} - \bar{G}'\|^2\right] \le \frac{|W| - K}{|W|\cdot K}\sigma^2 \tag{26}$$

If the global gradient is estimated based on the results of K workers, the corresponding variance can be bounded to:

$$\mathbb{E}\left[\|\bar{G}' - \nabla F(x)\|^2\right] \le \frac{1}{K}\sigma^2 \tag{27}$$

We then prove that the DNN model converges to a critical point when there are K_t workers participate in epoch t.

Lemma 8: Let $K_m = \min_{t\in\{1,\ldots,T\}} K_t$ and x_* denote the optimal model. Under Assumption 1, if the PS aggregates gradients from random K_t workers in epoch t with learning rate $\eta \le \frac{1}{L}$, the convergence result is as follows:

$$\frac{1}{T}\sum_{t=0}^{T-1}\mathbb{E}\left[\|\nabla F(x_t)\|^2\right] \le \frac{2(F(x_0) - F(x_*))}{\eta T} + \frac{\sigma^2}{K_m} \tag{28}$$

Proof: By the smoothness property in Assumption 1 and the definition of aggregated gradients $\bar{G}_t$, we have:

$$\begin{aligned}
&\mathbb{E}\left[F(x_{t+1}) - F(x_t)\right] \\
&\quad\le \langle \nabla F(x_t), \mathbb{E}\left[x_{t+1} - x_t\right]\rangle + \frac{L}{2}\mathbb{E}\left[\|x_{t+1} - x_t\|^2\right] \\
&\quad= -\eta\left\langle F(x_t), \mathbb{E}\left[\bar{G}_t\right]\right\rangle + \frac{L\eta^2}{2}\mathbb{E}\left[\|\bar{G}_t\|^2\right] \\
&\quad= -\eta\|F(x_t)\|^2 + \frac{L\eta^2}{2}\|\nabla F(x_t)\|^2 \\
&\qquad+ \frac{L\eta^2}{2}\mathbb{E}\left[\|\bar{G}_t - \nabla F(x_t)\|^2\right]
\end{aligned} \tag{29}$$

According to Lemma 7, we have:

$$\begin{aligned}
&\mathbb{E}\left[F(x_{t+1}) - F(x_t)\right] \\
&\qquad\le -\eta(1 - \frac{L\eta}{2})\|\nabla F(x_t)\|^2 + \frac{L\eta^2\sigma^2}{2K_t}
\end{aligned} \tag{30}$$

Rearrange the terms, we have

$$\begin{aligned}
&\eta(1 - \frac{L\eta}{2})\mathbb{E}\left[\|\nabla F(x_t)\|^2\right] \\
&\qquad\le \mathbb{E}\left[F(x_t) - f(x_{t+1})\right] + \frac{L\eta^2\sigma^2}{2K_t}
\end{aligned} \tag{31}$$

Given the fact that $\frac{1}{K_t} \le \frac{1}{K_m}$, we have

$$\begin{aligned}
&\mathbb{E}\left[F(x_t) - f(x_{t+1})\right] + \frac{L\eta^2\sigma^2}{2K_t} \\
&\qquad\le \mathbb{E}\left[F(x_t) - f(x_{t+1})\right] + \frac{L\eta^2\sigma^2}{2K_m}
\end{aligned} \tag{32}$$

Summing from $t = 0, \ldots, T-1$, we have the convergence result as following:

$$\begin{aligned}
&\frac{1}{T}\sum_{t=0}^{T-1}\mathbb{E}\left[\nabla\|F(x_t)\|^2\right] \\
&\quad\le \frac{\sum_{t=0}^{T-1}\mathbb{E}\left[F(x_t) - f(x_{t+1})\right]}{\eta T(1 - \frac{L\eta}{2})} + \frac{L\eta\sigma^2}{(2 - L\eta)K_m} \\
&\quad\le \frac{2(f(x_0) - f(x_*))}{\eta T} + \frac{\sigma^2}{K_m}
\end{aligned} \tag{33}$$

which completes the proof. ■

Theorem 9: Eq. (20) guarantees that the PS will receive K_t gradients from programmable switches, and the DNN model will converge to a critical point.

Proof: With collaborative in-network aggregation, each programmable switch is responsible for aggregating the sub-model gradients of workers. According to the fourth set of equations in Eq. (20), we can guarantee that the participating worker's all sub-model gradients will be received by the PS.

Therefore, the PS will receive at least K_t gradients in each epoch. Based on the Lemma 8, we can prove that the DNN model will be converged after T epochs. ■

From Theorem 9 we observe that we can exhibit better convergence performance as both K_m and T become larger. However, increasing the values of both K_m and T will lead to long training time. Thus, it is challenging to determine the optimal value of K_m and T to maintain a similar convergence performance with acceptable training time.

C. Algorithm Design

We first show that the number of epochs in the distributed training task is related to the completion time of each epoch by approximating the problem. Specifically, we set the upper bound of the mean square gradient to be less than a certain threshold (*e.g.*, $\epsilon \geq 0$), which is equivalent to meeting $F(x_T) \leq \mathcal{F}$. Subsequently, we have:

$$\frac{2(F(x_0)-F(x_*))}{\sqrt{nT}}+\frac{\sigma^2}{K_m}\leq \epsilon \tag{34}$$

Then, we rearrange the items and obtain:

$$T \geq \frac{4K_m^2(F(x_0)-F(x_*))^2}{n(\epsilon K_m-\sigma^2)^2} \tag{35}$$

Here, we define:

$$\psi(K_m)=\frac{4K_m^2(F(x_0)-F(x_*))^2}{n(\epsilon K_m-\sigma^2)^2} \tag{36}$$

To maintain a similar convergence performance and build the relationship between K_m and T, we set $T=\psi(K_m)$. Therefore, the objective can be reformulated as follows:

$$\min \sum_{t=1}^{\psi(K_m)} \bar{H} \tag{37}$$

where $\bar{H}$ represents the average completion time of each epoch. With $K_m=\min_{t\in\{1,\dots,T\}} K_t$, we approximate the problem of Eq. (20) as:

$$\min \sum_{t=1}^{\psi(K_m)} \bar{H}$$
$$S.t. \begin{cases} F(x_T)\leq \mathcal{F} & \\ H(t)=\max\{z_w^t\cdot H(w,g,t)\}+H(\alpha), & \forall t\in[T] \\ K_t=\sum\limits_{w\in W} z_w^t, & \forall t\in[T] \\ K_t^s=\sum\limits_{w\in W}\sum\limits_{g\in G} y_{w,g}^s\cdot z_w^t, & \forall t,s \\ z_w^t\in\{0,1\}, & \forall t,w \end{cases} \tag{38}$$

To decide the optimal value of K_t, we present a greedy-based algorithm in Alg. 2. We adopt the history gradient arrival time of workers to estimate the future gradient arrival time and the completion time of each epoch. By searching the candidate $K_t\in\{1,2,\dots,n\}$, we find the optimal K_t to minimize the total training time.

Specifically, for the PS, we use p_w to denote the history interval between distributing the parameters and receiving the last gradient from worker w, and p'_w to represent the estimated interval between the current moment and receiving the worker w's updated gradient. Let $\mathcal{P}=\{p_1,p_2,\dots,p_w\}$ and $\mathcal{P}'=\{p'_1,p'_2,\dots,p'_w\}$ represent the set of history and estimated intervals of workers, respectively. We sort the elements in $\mathcal{P}$ in ascending order, representing the interval of workers is increasing.

Algorithm 2 Online Searching for the Optimal Number of Participating Workers

```
1:  $H_{min} \leftarrow +\infty$
2:  for each $K \in \{1,2,3,\dots,|W|\}$ do
3:      for each $m \in \{1,2,3,..,M\}$ do
4:          Set $h_t \leftarrow p_{(K_m)}$
5:          for each $w \in W$ do
6:              if $p_w \geq h_t$ then
7:                  $p'_w \leftarrow p_w - h_t$
8:              else
9:                  $p'_w \leftarrow p_w$
10:             end if
11:         end for
12:         Set $\bar{H} \leftarrow \frac{\sum_{0\leq t'<t} h_{t'}}{t}$
13:     end for
14:     Set $T \leftarrow \psi(K_m)$
15:     Set $H \leftarrow T\cdot\bar{H}$
16:     if $H \leq H_{min}$ then
17:         Update $H_{min} \leftarrow H$
18:         Set $K_{t+1} \leftarrow K$
19:     end if
20: end for
21: Return $K_{t+1}$
```

For each candidate $K\in\{1,2,\dots,n\}$, we first initiate the estimated interval p'_w of worker w as its history interval p_w. Then we run M rounds to simulate the completion time of the following M epochs and obtain an average completion time $\bar{H}$. In practice, we can set M as the number of workers to ensure fast execution of our searching method. In each round, we set the completion time of epoch t as the estimated interval of K_m-th fastest worker (*i.e.*, $h_t\leftarrow p(K_m)$) to simulate that the PS aggregates gradients from K_m workers. If the estimated interval of worker w is greater than the completion time of the epoch, we reduce the estimated interval by h_t, which means the gradient of worker w will take additional time (*i.e.*, p_w-h_t) to arrive at the PS. Once we obtain the average completion time $\bar{H}$ of the epoch, we can compute the value of epochs T with selected K_m according to Eq. (36). Finally, we determine the optimal value of K_{t+1} by K_m that achieves the minimum value of training time $H=T\cdot\bar{H}$.

VI. Implementation

We implement the KRGS algorithm in the control plane and calculate the gradient scheduling policy via PuLP [46], where the sub-model set is defined by the default model partition of PyTorch [48]. We publish the policy by installing the corresponding entries to programmable switches with Barefoot Runtime Interface (BRI). Besides, for the asynchronous training, the PS will launch another thread to publish the value of K_t^s to each programmable switch via BRI. To simulate the network dynamics, the PS generates delayed latency and publishes them to workers via Secure Shell (SSH) connection before each training epoch. In the data plane, we implement the collaborative in-network aggregation with 583 LoCs of P4 in programmable switches. Specifically, we use 16384 registers for gradient aggregation and 2 match-action tables (MATs) for gradient filtering. We run PyTorch on each server to perform DT tasks. We obtain the parameter array of models by invoking the parameters_to_vector of PyTorch and partition the array into a set of gradient fragments. In the communication backend, we encode the gradient fragments into self-defined packet headers for in-network processing. Similar to [6] and [7], each gradient fragment contains 64 elements.

VII. Performance Evaluation

In this section, we evaluate GOAT with testbed and simulation experiments and highlight our findings as follows:

- GOAT speeds up distributed training by up to $1.77\times$ compared with existing solutions when performing practical DT tasks under network dynamics (Exp#1).
- GOAT reduces per epoch time by up to 52.7% (Exp#2) and the aggregation overhead of the PS by 93.8% (Exp#3) compared with other alternatives.
- By applying to the asynchronous training scenario, GOAT improves the training throughput by 49.7% (Exp#4) and speeds up the task by $2.02\times$ (Exp#5), compared with the synchronous scenario.
- GOAT offloads most of the gradient aggregation into programmable switches, increasing the in-network aggregation amount by up to $4\times$ compared with benchmarks (Exp#6).
- GOAT reduces the communication overhead by 31.1%-63.1% compared with the state-of-the-art solutions when encountering network dynamics (Exp#7).

A. Experimental Setup

Metrics. We adopt the following metrics for performance comparison: (1) training throughput; (2) test accuracy; (3) per epoch time; (4) communication time; (5) aggregation overhead of the PS; (6) in-network aggregation amount; and (7) communication overhead.

We measure the number of processed samples (*e.g.*, images) per second as *training throughput* and compute the ratio between the number of the samples correctly predicted by the model to the number of all samples in the test set as *test accuracy*. Then, we record the average duration between two consecutive epochs as *per epoch time*. In each epoch, we measure the average duration from the time a worker sends the gradient until the time that the worker finishes receiving the updated model as *communication time*. Besides, we use iftop [56] to monitor the total traffic amount of the PS, denoted as *aggregation overhead of the PS*. We calculate the size of gradients aggregated in programmable switches by subtracting the aggregation overhead of the PS from the total size of models as *in-network aggregation amount*. We sum the traffic size of gradients through links as *communication overhead*.

Benchmarks. We compare GOAT with four benchmarks. The first benchmark is a communication scheduling scheme without considering in-network aggregation, called Geryon [57]. Geryon computes the shortest path from each worker to the PS under resource constraints to transfer gradients. The second one, called ATP [6], performs in-network aggregation at top-of-rack programmable switches. For fairness, we let each worker sends the gradient to the PS via the shortest paths, where the gradient is aggregated in the first encountered aggregation node with available memory capacity. The third solution is the latest INA solution, called ESA [16]. It designs a priority-based preemption mechanism for asynchronously arriving gradients, where a gradient fragment with a high priority will evict the low priority fragment at the memory. For the asynchronous distributed training scenario (Section VII-C), we adopt a typical solution named K-SGD [53], as the fourth benchmark. K-SGD collects gradients from K fastest workers and discard stragglers in each epoch, to alleviate the influence of stragglers.

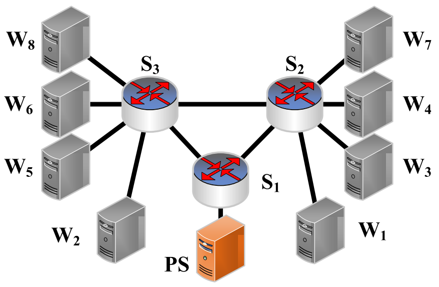


Fig. 4. Topology of the testbed consisting of 1 PS, 8 workers (W_1-W_8) and 3 programmable switches (S_1-S_3). All the components are connected via 100 Gbps links.

B. Testbed Settings and Results

Settings. We build the testbed with 3 Wedge100BF-32x programmable switches and 9 servers, as shown in Figure 4. The testbed topology is similar to the examples in Fig. 2(a), where 1 switch connects with 1 server hosting the PS and the other 2 switches connect with 4 servers hosting workers, respectively. Specifically, each switch features an Intel Tofino chip with Software Development Environment 9.3.1 and has the available memory of ∽20MB [13]. Each server has an NVIDIA GeForce RTX 3090, a 22-core Intel Xeon 6152 processor, and a Mellanox ConnectX-6 100G dual-port NIC. All servers run Ubuntu 18.04 with CUDA 11.3 and install the NIC driver with Mellanox driver OFED 5.5-1.0.3.2. Moreover, all programmable switches and servers are connected via 100Gbps links.

Workloads. We train two DNN models [2]: ResNet-18 with a size of 44MB and ResNet-50 with a size of 98MB on the Cifar-100 dataset [58]. Specifically, the dataset contains 60000 images (50000 for training and 10000 for testing), labeled in 100 classes. Similar to [7], we set the batch size to 64 and perform 200 training epochs for each DT task by default. Each programmable switch has the available memory of ∽20MB [13]. As a result, it needs 3 switches to collaboratively aggregate gradients of ResNet-18 and 5 switches to aggregate gradients of ResNet-50.

(Exp#1) Overall training performance. We measure the overall performance of DT tasks by evaluating the training throughput and test accuracy. The evaluation results are shown in Figs. 5-7. In Fig. 5, we set the number of workers to

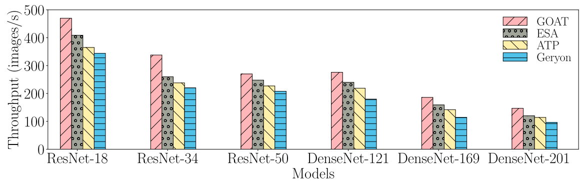


Fig. 5. Training Throughput vs. Models.

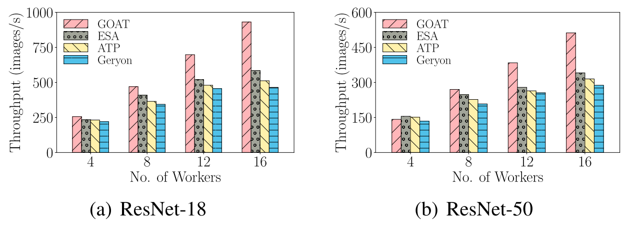


Fig. 6. Training Throughput vs. No. of Workers.

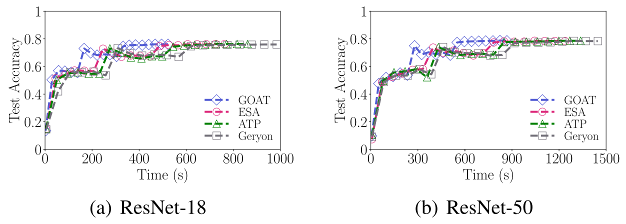


Fig. 7. Test Accuracy vs. Training Time.

8 and run several popular models: ResNet-18, ResNet-34, ResNet-50, DenseNet-121, DenseNet-169 and DenseNet-201 [2], [59]. The experimental results show that GOAT can obtain the highest training throughput among these benchmarks. For example, GOAT achieves a throughput of 276 images/s on average when training DenseNet-121, while ESA, ATP and Geryon obtain throughputs of 240 images/s, 219 images/s and 180 images/s, respectively. To save space, we only conduct a detailed performance comparison of all solutions with ResNet-18 and ResNet-50 in the following. In Fig. 6, we increase the number of workers from 4 to 16 with an increment of 4. The experimental results show that GOAT always achieves the highest training throughput as the number of workers increases. For example, given 16 workers in Fig. 6(a), the training throughputs of GOAT, ESA, ATP and Geryon are 931, 585, 512 and 465 images/s, respectively. In Fig. 7, we further record the test accuracy versus training time for the DT task containing 16 workers. It shows that GOAT takes the least time to complete the training task and achieves a similar test accuracy compared with other alternatives. For example, given ResNet-50 in Fig. 7(b), GOAT first reaches an accuracy of 0.7896 in 802.4s while the times of ESA, ATP and Geryon are 1208.48s, 1312.6s and 1416.9s, respectively. The results show that GOAT can speed up distributed training by 1.34×, 1.39× and 1.77×, compared with ESA, ATP and Geryon, respectively. The reason is that GOAT can aggregate more gradients in programmable switches through collaborative in-network aggregation, reducing the total communication overhead.

(Exp#2) Comparison on training time. This set of evaluations compares the training time performance of different solutions by varying the number of workers. Fig. 8 shows that, as the number of workers increases, the per epoch time increases too. Under the fixed number of workers, GOAT obtains the least per epoch time among all solutions. Given

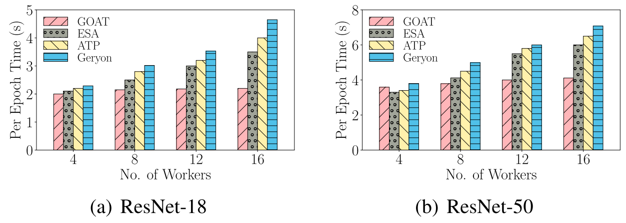


Fig. 8. Per Epoch Time vs. No. of Workers.

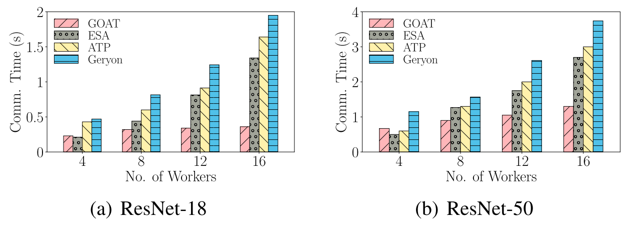


Fig. 9. Communication Time vs. No. of Workers.

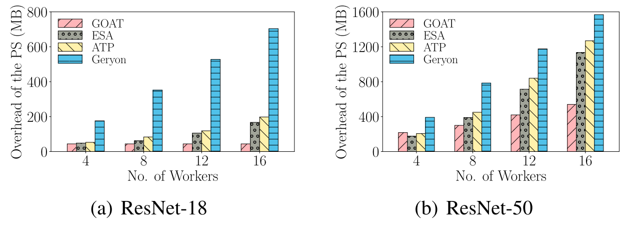
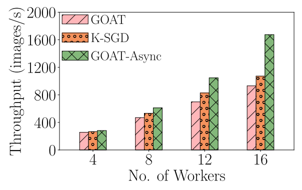


Fig. 10. Aggregation Overhead of the PS vs. No. of Workers.

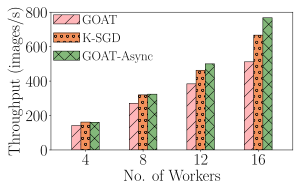
16 workers in Fig. 8(a), the per epoch times of GOAT, ESA, ATP and Geryon are 2.2s, 3.5s, 4s and 4.65s, respectively. We further estimate the communication time of each epoch. In Fig. 9(a), when the number of workers is 16, the communication times of GOAT, ESA, ATP and Geryon are 0.36s, 1.34s, 1.64s and 1.95s, respectively. Thus, by decreasing the communication time, GOAT reduces per epoch time by 37.14%, 45% and 52.7% compared with ESA, ATP and Geryon, respectively. Note that, each epoch consists of local training, communication and global aggregation. Our method does not optimize the local training time but can co-exist with solutions decreasing local training time if needed.

(Exp#3) Comparison on aggregation overhead of the PS. This set of evaluations estimates the average aggregation overhead of the PS in each epoch. Fig. 10 shows that GOAT can consistently achieve the least aggregation overhead of the PS compared with other alternatives. For example, given 16 workers in Fig. 10(a), the aggregation overheads of the PS of GOAT, ESA, ATP and Geryon are 44MB, 167.2MB, 198MB and 704MB, respectively. As a result, GOAT reduces the aggregation overhead of the PS by 73.6%, 77.8% and 93.8% compared with other benchmarks. The reason is that in Geryon all workers' gradients are sent to the PS without in-network aggregation. Besides, we find that workers' gradient packets arrive at switches asynchronously in real DT tasks. According to experimental results, in a task with 16 workers, the asynchronicity of the PS receiving gradients can reach up to 1.83s. Therefore, many asynchronously arriving gradients are sent to the PS without in-network aggregation in ESA and ATP, incurring higher aggregation overhead of the PS compared with GOAT.

Summary. Through collaboratively in-network aggregation, GOAT can achieve the highest training throughput, the

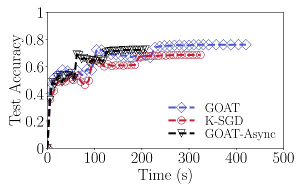


(a) ResNet-18

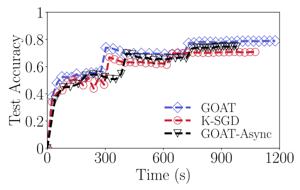


(b) ResNet-50

Fig. 11. Training Throughput vs. No. of Workers.



(a) ResNet-18



(b) ResNet-50

Fig. 12. Test Accuracy vs. Training Time.

least training time, and the least aggregation overhead of the PS compared with other benchmarks.

C. Asynchronous Training Settings and Results

Settings. We run asynchronous distributed training tasks with different numbers of workers, and compare the performance of GOAT under the normal scenario and the asynchronous training scenario (GOAT-Async). By default, we set the number of workers to 8 and run each distributed training task for 200 epochs. We vary the number of workers to 4, 8, 12, 16, separately. Similar to the work [60], we use normal distribution probability to simulate the existence of stragglers. Specifically, we randomly add $ratio \in [0, 1]$ seconds latency to all workers before sending gradients, where $ratio$ follows the normal distribution. We set the standard deviation of the normal distribution to 0.2. For each asynchronous training task, we run Alg. 2 to obtain the number of participations, and the PS will send the number of per-switch participations to the corresponding switch in each epoch. The PS only distributes the updated model to participating workers of the epoch. Note that, our algorithm can cooperate with other asynchronous training optimization frameworks [55], since we only decide the number of participating workers for each switch and do not involve the local training and global aggregation steps.

(Exp#4) Comparison on training throughput. We measure the training throughput of DT tasks under different numbers of workers (4)-(16). The evaluation results are shown in Fig. 11. It shows that GOAT-Async can achieve higher training throughput as the number of workers increases. For example, given 16 workers in Fig. 11(b), the training throughputs of GOAT, K-SGD and GOAT-Async are 512, 668 and 766 images/s, respectively. GOAT-Async increases training throughput by 49.7% and 14.7%, compared with GOAT and K-SGD, respectively. The reason is that GOAT-Async determines the number of participated workers in each epoch, reducing per-epoch time, and thereby increasing the training throughput.

(Exp#5) Comparison on time-to-accuracy. In Fig. 12, we record the test accuracy versus training time for the DT tasks containing 16 workers. It shows that GOAT-Async takes the least time to complete the training task among three solutions with a litter sacrifice in test accuracy compared with GOAT. Fig. 12(a) shows that, given 200 training epochs, GOAT-Async takes 207.7s to complete, while K-SGD and GOAT cost 324s and 420.4s, respectively. The results show that, GOAT-Async can speed up distributed training by 2.02× and 1.29×, compared with K-SGD and GOAT. Although the final test accuracy of GOAT is greater than that of GOAT-Async ($\approx 5\%$ improvement), GOAT-Async can achieve greater test accuracy than GOAT within the same training duration. For example, in Fig. 12(a), when the time is 200s, GOAT-Async's test accuracy is 0.7252, while that of GOAT is only 0.6856. The test accuracy of GOAT-Async outperforms that of GOAT by 5.78%. The reason is that GOAT-Async will calculate the optimal number of participations in each epoch, therefore the test accuracy will be improved faster than the synchronized training.

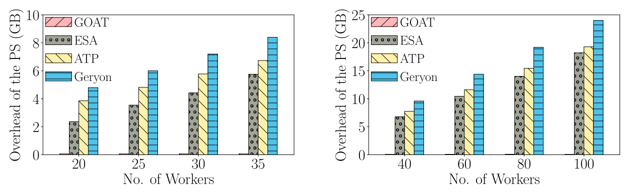
Note that, Fig. 12(b) shows some exceptions that the DNN model in GOAT convergeces faster than in GOAT-Async when the time is 310s. We think the reason is that ResNet-50 has more parameters than ResNet-18, increasing the complexity of convergence, which takes more epochs for GOAT-Async to train the DNN model to achieve higher test accuracy.

Summary. By calculating the optimal number of participating workers and deciding per-switch participations, GOAT can be applied to asynchronous training and speed up the distributed training tasks. Although the final model accuracy in GOAT-Async is smaller than that in GOAT, GOAT-Async can achieve faster model convergence given the same training time.

D. Simulation Settings and Results

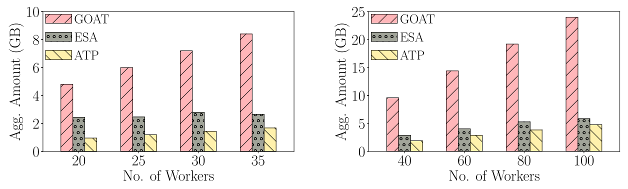
Settings. Our simulations are implemented on a physical server equipped with an Intel Core i9-10900 processor and 64GB RAM. Similar to Sec. VII-B, we adopt the LP solver PuLP [46] to compute the gradient scheduling policy. To verify the theoretical performance of GOAT, we select two practical topologies. The first topology is a leaf-spine topology [61], which consists of 20 switches (10 spine switches and 10 leaf switches) and 50 servers. The second one is a fat-tree topology [49], which contains 80 switches (32 edge switches, 32 aggregation switches and 16 core switches) and 192 servers. Considering the practical situation, we randomly select 20% of the switches as programmable switches with a memory of 64MB (same as the memory size of Intel Tofino 2). Each worker sends the traffic of 221MB (same as the gradient size of AlexNet [62]) to the PS. To simulate network dynamics, we set sending rates of workers with a normal distribution probability, similar to the work [60]. Specifically, we let the average sending rate of workers be 10Gbps. For each worker, we set its sending rate as $ratio \times 10$Gbps, where $ratio \in [0, 1]$ is obtained by the probability of normal distribution. By default, we set the standard deviation of the normal distribution to 0.2.

Note that, although network emulator mininet [63] supports replacing normal switches with software P4 switches (*i.e.*, bmv2 [64] switches) to simulate the network, it faces a



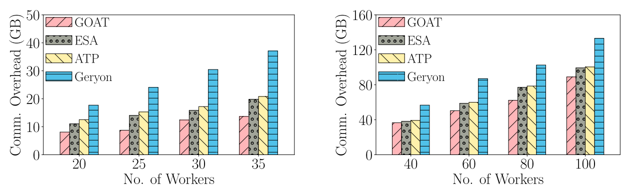
(a) Leaf-Spine (b) Fat-Tree

Fig. 13. Aggregation Overhead of the PS vs. No. of Workers.



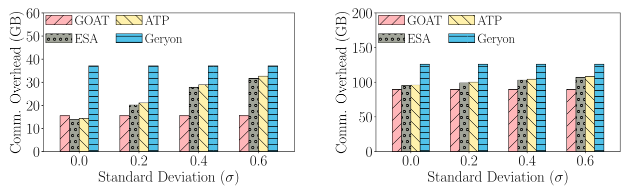
(a) Leaf-Spine (b) Fat-Tree

Fig. 14. In-network Aggregation Amount vs. No. of Workers.



(a) Leaf-Spine (b) Fat-Tree

Fig. 15. Communication Overhead vs. No. of Workers.



(a) Leaf-Spine (b) Fat-Tree

Fig. 16. Communication Overhead vs. Standard Deviations.

critical performance problem. Specifically, when the scale of topologies increases to tens of hosts, the bandwidth of bmv2 switches will degrade to several Mbps with high packet loss rates. The experimental results of the work [65] also confirmed this conclusion. Therefore, we do not choose to perform large-scale simulations through bmv2 and mininet, but through running the algorithm simulations.

(Exp#6) Comparison on gradient aggregation amount. We measure the gradient aggregation amount of four solutions, and the results are shown in Figs. 13-14. Fig. 13 shows that GOAT obtains the least aggregation overhead of the PS compared with other alternatives. For example, in the leaf-spine topology with 35 workers, the PS aggregation overheads of GOAT, ESA, ATP and Geryon are 60MB, 5.7GB, 6.7GB and 8.4GB, respectively. GOAT reduces the aggregation overhead of the PS by 98.9%, 99.1% and 99.2%, compared with ESA, ATP and Geryon, respectively.

Then, we consider the traffic aggregated by programmable switches (*i.e.*, in-network aggregation amount). Since Geryon does not perform in-network aggregation, we omit it in Fig. 14. We can see that GOAT achieves the highest in-network aggregation amount compared with other alternatives in Fig. 14. For example, in the fat-tree topology, given 40 workers, the in-network aggregation amounts of GOAT, ESA and ATP are 9.6GB, 3.2GB and 1.9GB, respectively. Our algorithm considers multiple switches to collaboratively perform in-network aggregation, moving the most traffic aggregated in programmable switches.

(Exp#7) Comparison on communication overhead. In this set of evaluations, we show the communication overhead of four solutions. Fig. 15 shows that GOAT achieves the least communication overhead compared with other benchmarks. Given 35 workers in the leaf-spine topology, the communication overheads of GOAT, ESA, ATP and Geryon are 13.7GB, 19.8GB, 20.9GB and 37.1GB, respectively. GOAT reduces communication overhead by 31.1%, 34.3% and 63.1% compared with ESA, ATP and Geryon, respectively. The reason is that GOAT schedules gradients to switches to minimize communication overhead.

In Fig. 16, we fix the number of workers and vary the normal distribution's standard deviation to evaluate the influence of network dynamics. As the degree of network dynamics increases, we can see that the communication overheads of ESA and ATP increase either. When the standard deviation is 0 (all workers have the same sending rates), ESA and ATP will aggregate all gradients in each worker's nearest programmable switches, thus gaining less communication overhead than GOAT. As the standard deviation increases, more and more traffic of ATP and ESA is aggregated in the PS, incurring massive communication overhead.

Summary. Through selecting optimal aggregation nodes for workers, GOAT can achieve the highest in-network aggregation amount and the least communication overhead compared with alternatives when encountering network dynamics.

VIII. Conclusion

In this paper, we present GOAT, a novel in-network aggregation approach with gradient scheduling. GOAT minimizes the communication overhead in the network by collaboratively conducting INA on multiple programmable switches. We further propose a knapsack-based randomized rounding algorithm for gradient scheduling and analyze its approximation performance. To speed up the distributed training, we extend GOAT to the asynchronous training scenario and present a greedy algorithm to determine the optimal number of participating workers in each epoch. Extensive testbed experimental and simulation results show that GOAT can efficiently aggregate asynchronously arriving gradients and accelerate the distributed training.

References

[1] J. Fang, G. Zhao, H. Xu, Z. Yu, B. Shen, and L. Xie, "GOAT: Gradient scheduling with collaborative in-network aggregation for distributed training," in *Proc. IEEE/ACM 31st Int. Symp. Quality Service (IWQoS)*, Jun. 2023, pp. 1–10.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2016, pp. 770–778.

[3] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," 2018, *arXiv:1810.04805*.

[4] S. Zhang, L. Yao, A. Sun, and Y. Tay, "Deep learning based recommender system: A survey and new perspectives," *ACM Comput. Surv.*, vol. 52, no. 1, pp. 1–38, Feb. 2019.

[5] M. Li et al., "Scaling distributed machine learning with the parameter server," in *Proc. Int. Conf. Big Data Sci. Comput.*, Aug. 2014, pp. 583–598.

[6] C. Lao et al., "ATP: In-network aggregation for multi-tenant learning," in *Proc. NSDI*, 2021, pp. 741–761.

[7] A. Sapio et al., "Scaling distributed machine learning with in-network aggregation," in *Proc. 18th USENIX Symp. Networked Syst. Design Implement. (NSDI)*, 2021, pp. 785–808.

[8] L. Luo, J. Nelson, L. Ceze, A. Phanishayee, and A. Krishnamurthy, "Parameter hub: A rack-scale parameter server for distributed deep neural network training," in *Proc. ACM Symp. Cloud Comput.*, Oct. 2018, pp. 41–54.

[9] J. Fei, C Y. Ho, A. N. Sahu, M. Canini, and A. Sapio, "Efficient sparse collective communication and its application to accelerate distributed deep learning," in *Proc. ACM SIGCOMM Conf.*, 2021, pp. 676–691.

[10] S. Grant, A. Yelam, M. Bland, and A. C. Snoeren, "SmartNIC performance isolation with FairNIC: Programmable networking for the cloud," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Architectures, Protocols Comput. Commun.*, Jul. 2020, pp. 681–693.

[11] Y. Li, I. Liu, Y. Yuan, D. Chen, A. Schwing, and J. Huang, "Accelerating distributed reinforcement learning with in-switch computing," in *Proc. ACM/IEEE 46th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2019, pp. 279–291.

[12] J. Fang, G. Zhao, H. Xu, C. Wu, and Z. Yu, "GRID: Gradient routing with in-network aggregation for distributed training," *IEEE/ACM Trans. Netw.*, vol. 31, no. 5, pp. 2267–2280, Oct. 2023.

[13] *Intel Tofino*. Accessed: Oct. 20, 2021. [Online]. Available: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html

[14] *32-Port Programmable Switch*. Accessed: Jun. 13, 2022. [Online]. Available: https://newwavedv.com/products/appliances/32-port-programmable-switch/

[15] D. Kim et al., "TEA: Enabling state-intensive network functions on programmable switches," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Architectures, Protocols Comput. Commun.*, Jul. 2020, pp. 90–106.

[16] H. Wang, Y. Qin, C. Lao, Y. Le, W. Wu, and K. Chen, "Efficient data-plane memory scheduling for in-network aggregation," 2022, *arXiv:2201.06398*.

[17] V. Addanki, O. Michel, and S. Schmid, "PowerTCP: Pushing the performance limits of datacenter networks," in *Proc. 19th USENIX Symp. Networked Syst. Design Implement.*, 2022, pp. 51–70.

[18] L. Zheng et al., "Alpa: Automating inter- and intra-operator parallelism for distributed deep learning," 2022, *arXiv:2201.12023*.

[19] Z. Shu et al., "Traffic engineering in software-defined networking: Measurement and management," *IEEE Access*, vol. 4, pp. 3246–3256, 2016.

[20] W. Xiao et al., "Gandiva: Introspective cluster scheduling for deep learning," in *Proc. 13th USENIX Conf. Operating Syst. Design Implement.*, 2018, pp. 595–610.

[21] S. Shalev-Shwartz and S. Ben-David, *Understanding Machine Learning: From Theory to Algorithms*. Cambridge, U.K.: Cambridge Univ. Press, 2014.

[22] J. Wang, H. Liang, and G. Joshi, "Overlap local-SGD: An algorithmic approach to hide communication delays in distributed SGD," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, 2020, pp. 8871–8875.

[23] J. Wang and G. Joshi, "Adaptive communication strategies to achieve the best error-runtime trade-off in local-update SGD," in *Proc. Mach. Learn. Syst.*, vol. 1, Apr. 2019, pp. 212–229.

[24] J. Wang and G. Joshi, "Cooperative SGD: A unified framework for the design and analysis of communication-efficient SGD algorithms," 2018, *arXiv:1808.07576*.

[25] Y. Li, M. Yu, S. Li, S. Avestimehr, N. S. Kim, and A. Schwing, "Pipe-SGD: A decentralized pipelined SGD framework for distributed deep net training," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 31, 2018, pp. 1–12.

[26] Z. Tang, S. Shi, X. Chu, W. Wang, and B. Li, "Communication-efficient distributed deep learning: A comprehensive survey," 2020, *arXiv:2003.06307*.

[27] S. Rajput, H. Wang, Z. Charles, and D. Papailiopoulos, "DETOX: A redundancy-based framework for faster and more robust gradient aggregation," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 1–11.

[28] L. Li, W. Xu, T. Chen, G. B. Giannakis, and Q. Ling, "RSA: Byzantine-robust stochastic aggregation methods for distributed learning from heterogeneous datasets," in *Proc. Conf. Artif. Intell. (AAAI)*, 2019, pp. 1544–1551.

[29] T. Chen, G. Giannakis, T. Sun, and W. Yin, "LAG: Lazily aggregated gradient for communication-efficient distributed learning," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 31, 2018, pp. 1–11.

[30] T. Vogels, S. P. Karimireddy, and M. Jaggi, "PowerSGD: Practical low-rank gradient compression for distributed optimization," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 1–10.

[31] M. Zinkevich, M. Weimer, L. Li, and A. Smola, "Parallelized stochastic gradient descent," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 23, 2010, pp. 1–9.

[32] R. L. Graham et al., "Scalable hierarchical aggregation protocol (SHArP): A hardware architecture for efficient data reduction," in *Proc. 1st Int. Workshop Commun. Optimizations HPC (COMHPC)*, Nov. 2016, pp. 1–10.

[33] H. Zhang, L. Stafman, A. Or, and M. J. Freedman, "SLAQ: Quality-driven scheduling for distributed machine learning," in *Proc. Symp. Cloud Comput.*, Sep. 2017, pp. 390–404.

[34] S. Liu et al., "In-network aggregation with transport transparency for distributed training," in *Proc. 28th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, vol. 3, 2023, pp. 376–391.

[35] H. Wang, Y. Qin, C. Lao, Y. Le, W. Wu, and K. Chen, "Preemptive switch memory usage to accelerate training jobs with shared in-network aggregation," in *Proc. IEEE 31st Int. Conf. Netw. Protocols (ICNP)*, Oct. 2023, pp. 1–12.

[36] Z. Li et al., "A2TP: Aggregator-aware in-network aggregation for multi-tenant learning," in *Proc. 18th Eur. Conf. Comput. Syst.*, 2023, pp. 639–653.

[37] J.-L. Ye, C. Chen, and Y. H. Chu, "A weighted ECMP load balancing scheme for data centers using P4 switches," in *Proc. IEEE 7th Int. Conf. Cloud Netw. (CloudNet)*, 2018, pp. 1–4.

[38] J. Liu, J. Huang, W. Lv, and J. Wang, "APS: Adaptive packet spraying to isolate mix-flows in data center network," *IEEE Trans. Cloud Comput.*, vol. 10, no. 2, pp. 1038–1051, Apr. 2022.

[39] H. Liang et al., "DNN surgery: Accelerating DNN inference on the edge through layer partitioning," *IEEE Trans. Cloud Comput.*, vol. 11, no. 3, pp. 3111–3125, Jul./Sep. 2023.

[40] T. Mohammed, C. Joe-Wong, R. Babbar, and M. D. Francesco, "Distributed inference acceleration with adaptive DNN partitioning and offloading," in *Proc. IEEE Conf. Comput. Commun.*, Jul. 2020, pp. 854–863.

[41] T. Alonso et al., "Elastic-DF: Scaling performance of DNN inference in FPGA clouds through automatic partitioning," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, no. 2, pp. 1–34, Jun. 2022.

[42] H. Li, X. Li, Q. Fan, Q. He, X. Wang, and V. C. M. Leung, "Distributed DNN inference with fine-grained model partitioning in mobile edge computing networks," *IEEE Trans. Mobile Comput.*, early access, pp. 1–15, Jan. 24, 2024.

[43] L. Lin, X. Liao, H. Jin, and P. Li, "Computation offloading toward edge computing," *Proc. IEEE*, vol. 107, no. 8, pp. 1584–1607, Aug. 2019.

[44] *Intel Tofino 2*. Accessed: Jun. 19, 2022. [Online]. Available: https://www.intel.cn/content/www/cn/zh/products/network-io/programmable-ethernet-switch/tofino-2-series.html

[45] W. Wen et al., "Terngrad: Ternary gradients to reduce communication in distributed deep learning," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 1–11.

[46] *Pulp*. Accessed: Jul. 20, 2021. [Online]. Available: https://pypi.org/project/PuLP/

[47] G. Zhao, H. Xu, S. Chen, L. Huang, and P. Wang, "Joint optimization of flow table and group table for default paths in SDNs," *IEEE/ACM Trans. Netw.*, vol. 26, no. 4, pp. 1837–1850, Aug. 2018.

[48] A. Paszke et al., "PyTorch: An imperative style, high-performance deep learning library," in *Proc. NIPS*, Dec. 2019, pp. 8024–8035.

[49] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 63–74, 2008.

[50] X. Su, X. Yan, and C.-L. Tsai, "Linear regression," *Wiley Interdiscipl. Rev., Comput. Statist.*, vol. 4, no. 3, pp. 275–294, 2012.

[51] M. P. LaValley, "Logistic regression," *Circulation*, vol. 117, no. 18, pp. 2395–2399, 2008.

[52] S. Suthaharan, "Support vector machine," in *Machine Learning Models and Algorithms for Big Data Classification: Thinking With Examples for Effective Learning*. Cham, Switzerland: Springer, 2016, pp. 207–235.

[53] S. Dutta, G. Joshi, S. Ghosh, P. Dube, and P. Nagpurkar, "Slow and stale gradients can win the race: Error-runtime trade-offs in distributed SGD," in *Proc. Int. Conf. Artif. Intell. Statist.*, 2018, pp. 803–812.

[54] Y. Xu, Y. Liao, H. Xu, Z. Ma, L. Wang, and J. Liu, "Adaptive control of local updating and model compression for efficient federated learning," *IEEE Trans. Mobile Comput.*, vol. 22, no. 10, pp. 5675–5689, Oct. 2023.

[55] J. Xu, S.-L. Huang, L. Song, and T. Lan, "Live gradient compensation for evading stragglers in distributed learning," in *Proc. IEEE Conf. Comput. Commun.*, May 2021, pp. 1–10, doi: 10.1109/INFOCOM42981.2021.9488815.

[56] *IFTOP*. Accessed: Apr. 14, 2022. [Online]. Available: http://www.ex-parrot.com/ pdw/iftop/

[57] S. Wang, D. Li, and J. Geng, "Geryon: Accelerating distributed CNN training by network-level flow scheduling," in *Proc. INFOCOM*, 2020, pp. 1678–1687.

[58] A. Krizhevsky, V. Nair, and G. Hinton. *CIFAR-100 (Canadian Institute for Advanced Research)*. Accessed: Apr. 21, 2024. [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html

[59] F. Iandola, M. Moskewicz, S. Karayev, R. Girshick, T. Darrell, and K. Keutzer, "DenseNet: Implementing efficient ConvNet descriptor pyramids," 2014, *arXiv:1404.1869*.

[60] V. Jalaparti, I. Bliznets, S. Kandula, B. Lucier, and I. Menache, "Dynamic pricing and traffic engineering for timely inter-datacenter transfers," in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 73–86.

[61] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella, "Presto: Edge-based load balancing for fast datacenter networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 465–478, 2015.

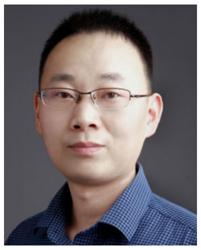
[62] O. Russakovsky et al., "ImageNet large scale visual recognition challenge," *Int. J. Comput. Vis.*, vol. 115, no. 3, pp. 211–252, Dec. 2015.

[63] *An Instant Virtual Network on Your Laptop*. Accessed: Apr. 21, 2024. [Online]. Available: http://mininet.org

[64] *Behavioral Model Version 2 (BMV2)*. Accessed: Apr. 21, 2024. [Online]. Available: https://github.com/p4lang/behavioral-model

[65] A. Sapio, I. Abdelaziz, M. Canini, and P. Kalnis, "DAIET: A system for data aggregation inside the network," in *Proc. Symp. Cloud Comput.*, Sep. 2017, p. 626.

Jin Fang received the B.S. degree from the College of Computer Science and Electronic Engineering, Hunan University, in 2020. He is currently pursuing the Ph.D. degree in computer software and theory with the University of Science and Technology of China. His current research interests include software-defined networks, distributed training systems, and programmable networks.

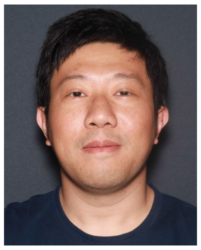


Hongli Xu (Member, IEEE) received the B.S. degree in computer science and the Ph.D. degree in computer software and theory from the University of Science and Technology of China (USTC), China, in 2002 and 2007, respectively. He is currently a Professor with the School of Computer Science and Technology, USTC. He has published more than 100 papers in famous journals and conferences, including IEEE/ACM TRANSACTIONS ON NETWORKING, IEEE TRANSACTIONS ON MOBILE COMPUTING, IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, International Conference on Computer Communications (INFOCOM), and International Conference on Network Protocols (ICNP). He has held more than 30 patents. His research interests include software-defined networks, edge computing, and the Internet of Things. He was awarded the Outstanding Youth Science Foundation of NSFC in 2018. He has won the best paper award or the best paper candidate at several famous conferences.

Gongming Zhao (Member, IEEE) received the Ph.D. degree in computer software and theory from the University of Science and Technology of China in 2020. He is currently an Associate Professor with the University of Science and Technology of China. His current research interests include software-defined networks and cloud computing.

Zhuolong Yu received the bachelor's and master's degrees from the University of Science and Technology of China and the Ph.D. degree from the Department of Computer Science, Johns Hopkins University. His research interests include networking systems, with a focus on programmable networks.

Bingchen Shen received the B.S. degree from Beijing University of Technology in 2022. He is currently pursuing the master's degree with the School of Computer Science and Technology, University of Science and Technology of China. His main research interests include data center networks.



Liguang Xie (Senior Member, IEEE) received the Ph.D. degree in computer engineering from Virginia Tech, Blacksburg, VA, USA, in 2013. He is currently a Senior Principal Architect and the Senior Director of engineering with Seattle Research Center, Futurewei Technologies Inc., where he leads the Cloud Networking Research and Development Team and oversees cloud networking open-source, research, and engineering efforts. Before Futurewei, he was a Senior Software Engineering Lead with Microsoft Azure Networking and an Adjunct Assistant Professor with the Bradley Department of Electrical and Computer Engineering, Virginia Tech. His research interests include cloud networking, software-defined networking, distributed systems, cloud computing, and AI systems.