GOAT: Gradient Scheduling with Collaborative In-Network Aggregation for Distributed Training

Jin Fang[1,2] Gongming Zhao[1,2] Hongli Xu[1,2] Zhuolong Yu[3] Bingchen Shen[1,2] Liguang Xie[4]
[1]School of Computer Science and Technology, University of Science and Technology of China
[2]Suzhou Institute for Advanced Research, University of Science and Technology of China
[3] Johns Hopkins University, USA
[4] Virginia Tech, USA

***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 the DT. 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**

I. Introduction

With the increasing complexity of machine learning (ML) applications, such as computer vision [1], natural language processing [2] and recommender systems [3], the scale of ML tasks is growing explosively. In practice, distributed training (DT) [4], 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 [5, 6, 7, 8]. For example, for a DT task training BERT on 10Gbps links, 67% of the training time is occupied for communication [6].

Triggered by the recent rise of programmable networking [9], in-network aggregation (INA) [5, 6, 10, 11] 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 [12]) 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 [10].

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 [6]. One intuitive solution is to increase the on-chip memory size directly. But it requires chip modification and raises the cost significantly. Alternatively, TEA [13] 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 [5, 6, 14] 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* [14]. For example, ATP [5] 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 arriving gradient fragments in the same memory unit. Instead, these fragments are forwarded directly to the PS, neglecting the benefit of in-network aggregation (see Sec. II-A for details).

To deal with this issue, some works [5, 6] 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 [5] 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 [15], 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 [16]. 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 [17] ineffective. Therefore, it is challenging to design an efficient gradient scheduling scheme with collaborative in-network aggregation to mitigate the communication bottleneck of DT. The main contributions of this paper are as follows:

1) We design GOAT, the first-of-its-kind work which 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 prove its NP-Hardness. We 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 can be tightened to* 4.
3) We prototype GOAT with 3 Intel Tofino switches. We use both testbed experiments and simulations to demonstrate the effectiveness of GOAT. The 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.

II. Motivation and GOAT Overview

A. Memory Sharing Scheme

Due to the limited size of switch memory, existing solutions [5, 6, 14] 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

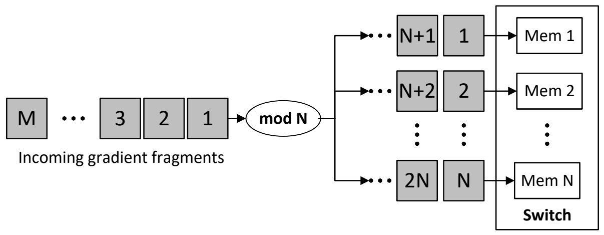


Fig. 1: Illustration of memory sharing scheme: the switch memory can be divided into N memory units. Correspondingly, a gradient can be partitioned into M gradient fragments, each assigned to a memory unit via hash function for aggregation. Since the size of switch memory is usually smaller than the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

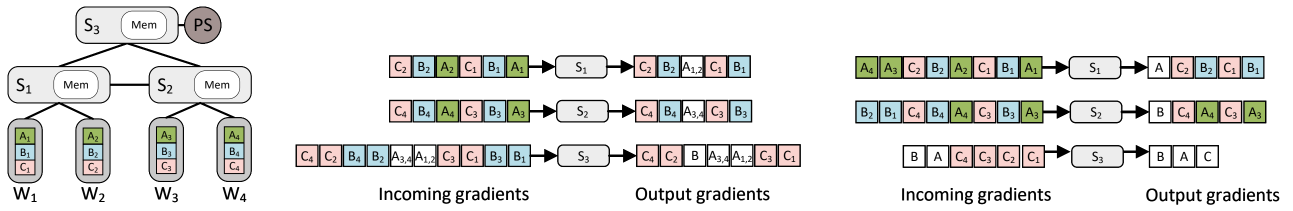
of gradients (*i.e.*, gradient fragments) at a time, as illustrated in Fig. 1. Correspondingly, the gradient is divided into M fragments, each having the same size as the memory unit. When one fragment arrives at a switch, it will be hashed to a specific unit according to its index (*e.g.*, fragment 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. Therefore, asynchronously arriving fragments may encounter hash collision, degrading the throughput of in-network aggregation.

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 [5]. 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 $A_{1,2}$. Then it aggregates B_1 with B_2 and outputs $B_{1,2}$. Finally



(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.

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, 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 can not 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, we argue that this method can not prevent massive traffic aggregated by the PS in time.

Since one programmable switch can not 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.

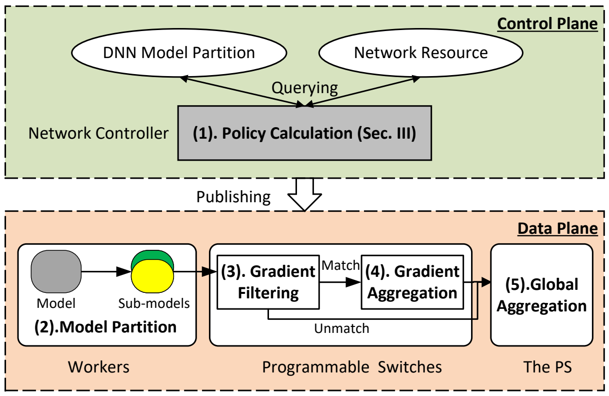


Fig. 3: System overview of GOAT. GOAT is composed of two parts. The control plane is responsible for determining the gradient scheduling policy. The data plane is responsible for collaborative in-network aggregation.

Discussion. The above example illustrates the idea of gradient scheduling with collaborative in-network aggregation. In practice, we can partition the model into sub-models with existing methods such as [18], 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 [1] is 98MB and the memory size of Intel Tofino 2 [19] is 64MB. So it only takes 2 switches to aggregate ResNet-50's gradients. For large models, we can co-exist with methods such as gradient quantization [20] 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 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 III. For the data plane, we can implement aggregation operations based on existing solutions [5, 6]. Due to space limitations, we omit the design details of the data plane and present the workflow in Sec. II-D.

D. Workflow of GOAT

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

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.

III. Problem Formulation and Algorithm Design

A. System Model

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

DNN Model Training. A DNN model is partitioned into a set of sub-models, whose gradient can be denoted as $G = \{g_1, g_2, \dots, g_{|G|}\}$. 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 $S = \{s_1, s_2, \dots, s_{|S|}\}$ 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 key step of GINA is determining to which aggregation nodes each worker should send its sub-model gradients. Thus, let $y^s_{w,g} \in \{0,1\}$ represent whether aggregation node s aggregates worker w's gradient g, or not. Let $x^s_g \in \{0,1\}$ represent whether sub-model gradient g is aggregated in aggregation node s, or not. The problem can be formulated as follows.

$$\min \sum_{g\in G}\Big(\sum_{w\in W}\sum_{s\in S\cup\{\alpha\}} y^s_{w,g}\cdot D_w(s) + \sum_{s\in S} x^s_g \cdot D_s(\alpha)\Big)\cdot b(g)$$
$$S.t.\begin{cases} \sum_{s\in S\cup\{\alpha\}} x^s_g \ge 1, & \forall g\in G\\ \sum_{s\in S\cup\{\alpha\}} y^s_{w,g} = 1, & \forall w\in W, g\in G\\ y^s_{w,g} \le x^s_g, & \forall w\in W, g\in G, s\in S\cup\{\alpha\}\\ \sum_{g\in G} x^s_g\cdot b(g)\le B(s), & \forall s\in S\\ x^s_g\in\{0,1\}, & \forall g\in G, s\in S\cup\{\alpha\}\\ y^s_{w,g}\in\{0,1\}, & \forall w\in W, g\in G, s\in S\cup\{\alpha\}\end{cases} \quad (1)$$

The first set of inequalities denotes that for each sub-model gradient, at least one aggregation node is responsible for aggregation. The second set of equations represents that for each worker, each sub-model gradient should be aggregated in one aggregation node. The third set of inequalities represents that each gradient can only be aggregated in the designated aggregation node. The fourth set of inequalities means the programmable switch memory constraint. 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.

Theorem 1: The GINA problem is NP-hard.

Proof: We prove the NP-hardness by showing that the generalized assignment problem [21] is a special case of GINA. Due to limited space, we omit it here. ■

C. Algorithm Design

Supposing that programmable switches do not aggregate gradients, the total size of transferred gradients can be calculated by

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

To minimize the total size of transferred gradients in the network, we need to maximize the traffic of aggregated gradients. By subtracting Eq. (2) from the objective of Eq. (1), we can obtain the traffic of aggregated gradients as follows:

$$\sum_{g\in G}\sum_{s\in S}(\sum_{w\in W} y^s_{w,g}-x^s_g)\cdot D_s(\alpha)\cdot b(g) \qquad (3)$$

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

$$\max \sum_{g\in G}\sum_{s\in S}(\sum_{w\in W} y^s_{w,g}-x^s_g)\cdot D_s(\alpha)\cdot b(g)$$

$$S.t.\begin{cases}\sum\limits_{s\in S\cup\{\alpha\}} x^s_g\geq 1, & \forall g\in G\\ \sum\limits_{s\in S\cup\{\alpha\}} y^s_{w,g}=1, & \forall w\in W, g\in G\\ y^s_{w,g}\leq x^s_g, & \forall w\in W, g\in G, s\in S\cup\{\alpha\}\\ \sum\limits_{g\in G} x^s_g\cdot b(g)\leq B(s), & \forall s\in S\\ x^s_g\in\{0,1\}, & \forall g\in G, s\in S\cup\{\alpha\}\\ y^s_{w,g}\in\{0,1\}, & \forall w\in W, g\in G, s\in S\cup\{\alpha\}\end{cases} \qquad (4)$$

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 $\{x^s_g, y^s_{w,g}\}$ with their fractional versions. We can solve it with a linear program solver (*e.g.*, PULP [22]) and the optimal solution is denoted as $\{\widetilde{x}^s_g,\widetilde{y}^s_{w,g}\}$. 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 $k(g)=\left\lfloor\sum\limits_{s\in S}\widetilde{x}^s_g\right\rfloor$, which is the required number of programmable switches to aggregate gradient g. Then we put variables $\widetilde{x}^s_g$ ($\forall g\in G$) into $k(g)$ knapsacks with min-max sum. For each knapsack a, sub-model gradient g will be scheduled to switch s with probability $\frac{\widetilde{x}^s_g}{\mathcal{S}_a}$, where $\mathcal{S}_a$ is the sum of x^s_g 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

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

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

of selecting switch $s\in S(g)$ to aggregate as $p_n(s)=\frac{\widetilde{y}^s_{w,g}}{\widetilde{x}^s_g}$ 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 2: 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.*, $\widetilde{x}^\alpha_g=0$. According to the first set of inequalities in Eq. (4) and the definition of $k(g)$, we have:

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

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

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

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

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 ($p^\alpha_n=1-\sum\limits_{s\in S(g)}p_n(s)=1$).

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

Theorem 3: 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 4: 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:

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

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 $\mathcal{X}_2 = \mathcal{X}_2 - \left\{x_m^1, x_m^2\right\}$. Otherwise, $\mathcal{X}_2 = \mathcal{X}_2 - \left\{x_m^1, x_m^2\right\} + x_m^3$. 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. ■

Lemma 5: **Chernoff Bound** [23]: Given n independent variables: $y_1, y_2, \dots, y_n, \forall y_i \in [0, 1]$. 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 6: 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 4, 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:

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

By applying Lemma 5, we have:

$$\begin{aligned} &\Pr\left[\sum_{g\in G} \frac{\delta_s \cdot \nu}{2 \cdot B_s} \ge (1+\varrho)\cdot\nu\right] \le e^{\frac{-\varrho^2\nu}{2+\varrho}} \\ \Rightarrow &\Pr\left[\sum_{g\in G} \frac{\delta_s}{2 \cdot B_s} \ge (1+\varrho)\right] \le e^{\frac{-\varrho^2\nu}{2+\varrho}} \end{aligned} \tag{15}$$

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

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

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

$$\begin{aligned} &\varrho \ge \frac{\log|S| + \sqrt{\log^2|S| + 8\nu\log|S|}}{2\nu}, (|S| \ge 2) \\ \Rightarrow &\varrho \ge \frac{\log|S|}{\nu} + 2, (|S| \ge 2) \end{aligned} \tag{17}$$

In practice, the on-chip memory size of Intel Tofino 2 is 64MB [19]. According to the default model partition of BERT in PyTorch [24], 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 $|S| = 20$, so $3 \cdot \log|S| \approx 3.9$. Combining these assumptions, we can obtain that $\nu \ge 3 \cdot \log|S|$. As a result, we have:

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

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.*, $\nu \ge 3 \cdot \log|S|$), the bound can be tightened to 4. ■

Theorem 7: 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 $\mathbb{E}\left[\widehat{y}^s_{w,g}\right] \le 2 \cdot \widetilde{y}^s_{w,g}$. 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 $\mathbb{E}\left[\widehat{y}^s_{w,g}\right] = \frac{\widetilde{y}^s_{w,g}}{\widetilde{x}^s_g} \cdot \frac{\widetilde{x}^s_g}{\mathcal{S}_a}$. According to Lemma 4, we can obtain $\mathbb{E}\left[\widehat{y}^s_{w,g}\right] = \frac{\widetilde{y}^s_{w,g}}{\mathcal{S}_a} \le 2 \cdot \widetilde{y}^s_{w,g}$. Then we can analyze the approximation ratio performance based on the randomized rounding method. Since the proof process is similar to that of Theorem 6 and the space is limited, we omit it here. ■

IV. Performance Evaluation

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 [25] 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 three benchmarks. The first benchmark is a communication scheduling scheme without considering in-network aggregation, called Geryon [26]. Geryon computes the shortest path from each worker to the PS under resource constraints to transfer gradients. The second one, called ATP [5], 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 last solution is the latest INA solution, called ESA [14]. 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.

B. Testbed Settings and Results

Settings. We build the testbed with 3 Wedge100BF-32x programmable switches and 9 servers. 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

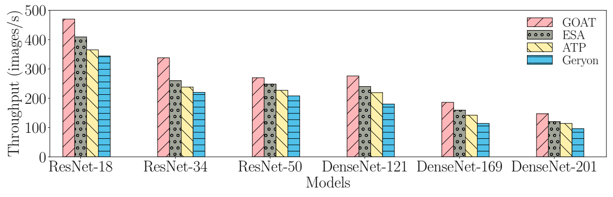


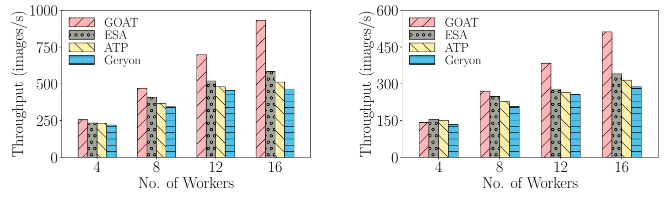
Fig. 4: Training Throughput vs. Models

∽20MB [12]. 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 [1]: ResNet-18 with a size of 44MB and ResNet-50 with a size of 98MB on the Cifar-100 dataset [27]. Specifically, the dataset contains 60000 images (50000 for training and 10000 for testing), labeled in 100 classes. Similar to [6], we set the batch size to 64 and perform 200 training epochs for each DT task by default.

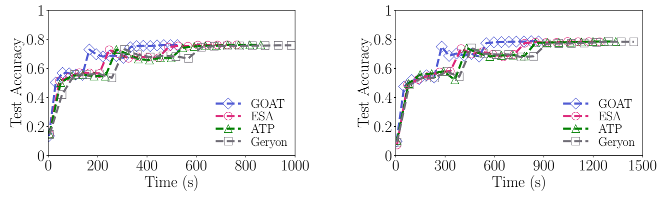
Implementation. We implement the KRGS algorithm in the control plane and calculate the gradient scheduling policy via PuLP [22], where the sub-model set is defined by the default model partition of PyTorch [24]. We publish the policy by installing the corresponding entries to programmable switches with Barefoot Runtime Interface (BRI). 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 [5, 6], each gradient fragment contains 64 elements.

(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. 4-6. In Fig. 4, we set the number of workers to 8 and run several popular models: ResNet-18, ResNet-34, ResNet-50, DenseNet-121, DenseNet-169 and DenseNet-201 [1, 28]. 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. 5, 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. 5(a), the training throughputs of GOAT, ESA, ATP and Geryon are 931,



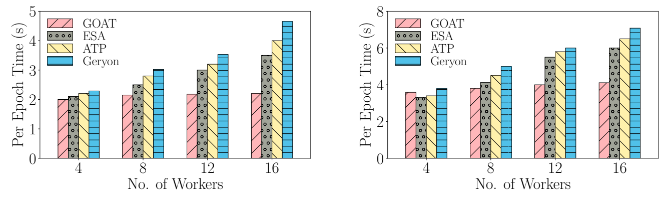
(a) ResNet-18 (b) ResNet-50

Fig. 5: Training Throughput vs. No. of Workers



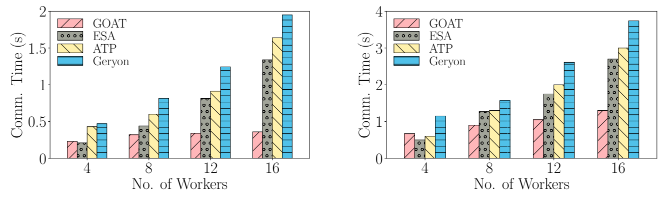
(a) ResNet-18 (b) ResNet-50

Fig. 6: Test Accuracy vs. Training Time



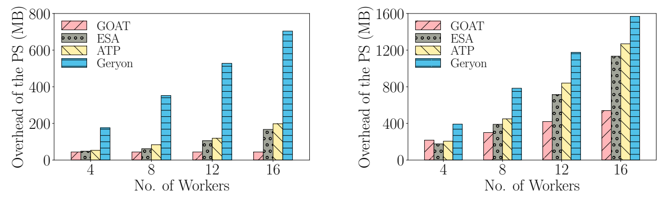
(a) ResNet-18 (b) ResNet-50

Fig. 7: Per Epoch Time vs. No. of Workers



(a) ResNet-18 (b) ResNet-50

Fig. 8: Communication Time vs. No. of Workers



(a) ResNet-18 (b) ResNet-50

Fig. 9: Aggregation Overhead of the PS vs. No. of Workers

585, 512 and 465 images/s, respectively. In Fig. 6, 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. 6(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 $1.34\times$, $1.39\times$ and $1.77\times$, 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.

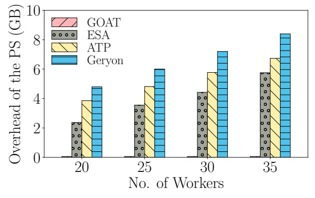
(Exp#2) Comparison on training time. This set of evaluations compares the training time performance of different solutions by varying the number of workers. Fig. 7 shows that, as the number of workers increases, the per epoch time increases too. Under the fixed number of workers, GOAT obtains the least per epoch time among all solutions. Given 16 workers in Fig. 7(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. 8(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. 9 shows that GOAT can consistently achieve the least aggregation overhead of the PS compared with other alternatives. For example, given 16 workers in Fig. 9(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 least training time, and the least aggregation overhead of the PS compared with other benchmarks.

C. 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. IV-B, we adopt the LP solver PuLP [22] 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 [29], which consists of 20 switches (10 spine switches and 10


(a) Leaf-Spine

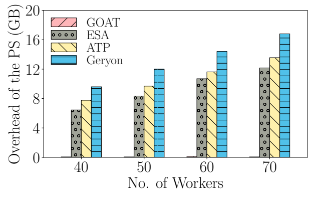
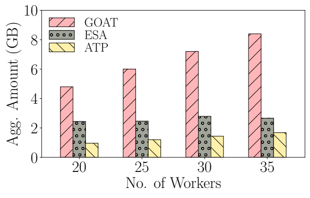

(b) Fat-Tree

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


(a) Leaf-Spine

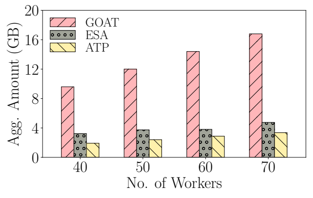
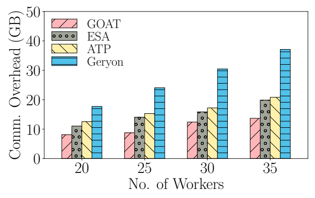

(b) Fat-Tree

Fig. 11: In-network Aggregation Amount vs. No. of Workers


(a) Leaf-Spine

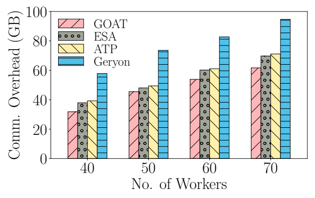
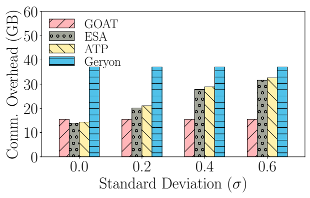

(b) Fat-Tree

Fig. 12: Communication Overhead vs. No. of Workers


(a) Leaf-Spine

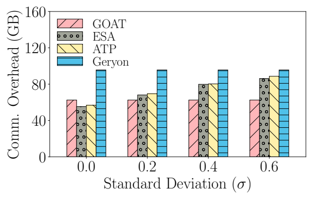

(b) Fat-Tree

Fig. 13: Communication Overhead vs. Standard Deviations

leaf switches) and 50 servers. The second one is a fat-tree topology [30], 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 [31]) to the PS. To simulate network dynamics, we set sending rates of workers with a normal distribution probability, similar to the work [32]. Specifically, we let the average sending rate of workers be 10Gbps. For each worker, we set its sending rate as $ratio \times$10Gbps, 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 [33] supports replacing normal switches with software P4 switches (*i.e.*, bmv2 [34] switches) to simulate the network, it faces a 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 [35] 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#4) Comparison on gradient aggregation amount. We measure the gradient aggregation amount of four solutions, and the results are shown in Figs. 10-11. Fig. 10 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. 11. We can see that GOAT achieves the highest in-network aggregation amount compared with other alternatives in Fig. 11. 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#5) Comparison on communication overhead. In this set of evaluations, we show the communication overhead of four solutions. Fig. 12 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. 13, 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.

V. 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. Extensive testbed experimental and simulation results show that GOAT can efficiently aggregate asynchronously arriving gradients and accelerate the distributed training.

Acknowledgement

The corresponding author of this paper is Gongming Zhao. This article was supported in part by the National Science Foundation of China (NSFC) under Grants 62102392 and 62132019; in part by the Open Research of Projects of Zhejiang Lab under Grant 2022QA0AB04; in part by the National Science Foundation of Jiangsu Province under Grant BK20210121; in part by the Fundamental Research Funds for the Central Universities; in part by the Youth Innovation Promotion Association of CAS (2023481).

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” *ArXiv*, vol. abs/1810.04805, 2019.

[3] S. Zhang, L. Yao, A. Sun, and Y. Tay, “Deep learning based recommender system: A survey and new perspectives,” *ACM Computing Surveys (CSUR)*, vol. 52, no. 1, pp. 1–38, 2019.

[4] M. Li, “Scaling distributed machine learning with the parameter server,” in *Proceedings of the 2014 International Conference on Big Data Science and Computing*.

[5] C. Lao, Y. Le *et al.*, “ATP: In-network aggregation for multi-tenant learning,” in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, Apr. 2021, pp. 741–761.

[6] A. Sapio, M. Canini *et al.*, “Scaling distributed machine learning with In-Network aggregation,” in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, Apr. 2021, pp. 785–808.

[7] L. Luo *et al.*, “Parameter hub: a rack-scale parameter server for distributed deep neural network training,” in *Proceedings of the ACM Symposium on Cloud Computing*, 2018.

[8] J. Fei *et al.*, “Efficient sparse collective communication and its application to accelerate distributed deep learning,” in *Proceedings of the ACM SIGCOMM 2021 Conference*.

[9] S. Grant, A. Yelam *et al.*, “Smartnic performance isolation with fairnic: Programmable networking for the cloud,” in *Proceedings of the ACM SIGCOMM 2020 Conference*, 2020.

[10] Y. Li *et al.*, “Accelerating distributed reinforcement learning with in-switch computing,” in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*.

[11] J. Fang, G. Zhao, H. Xu, C. Wu, and Z. Yu, “Grid: Gradient routing with in-network aggregation for distributed training,” *IEEE/ACM Transactions on Networking*, pp. 1–14, 2023.

[12] 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

[13] D. Kim, Z. Liu *et al.*, “Tea: Enabling state-intensive network functions on programmable switches,” in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication*, 2020.

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

[15] V. Addanki, O. Michel, and S. Schmid, “{PowerTCP}: Pushing the performance limits of datacenter networks,” in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 51–70.

[16] L. Zheng, Z. Li *et al.*, “Alpa: Automating inter-and intra-operator parallelism for distributed deep learning,” *arXiv preprint arXiv:2201.12023*, 2022.

[17] Z. Shu *et al.*, “Traffic engineering in software-defined networking: Measurement and management,” *IEEE access*, 2016.

[18] T. Alonso, L. Petrica *et al.*, “Elastic-df: Scaling performance of dnn inference in fpga clouds through automatic partitioning,” *ACM Trans. Reconfigurable Technol. Syst.*, 2021.

[19] 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

[20] W. Wen, C. Xu *et al.*, “Terngrad: Ternary gradients to reduce communication in distributed deep learning,” 2017.

[21] D. B. Shmoys and É. Tardos, “An approximation algorithm for the generalized assignment problem,” *Mathematical programming*, vol. 62, no. 1, pp. 461–474, 1993.

[22] Pulp. Accessed: July. 20, 2021. [Online]. Available: https://pypi.org/project/PuLP/

[23] 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 Transactions on Networking*, 2018.

[24] A. Paszke *et al.*, “Pytorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems 32*, 2019, pp. 8024–8035.

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

[26] S. Wang, D. Li, and J. Geng, “Geryon: Accelerating distributed cnn training by network-level flow scheduling,” in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2020, pp. 1678–1687.

[27] A. Krizhevsky, V. Nair, and G. Hinton, “Cifar-100 (canadian institute for advanced research).” [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html

[28] F. Iandola, M. Moskewicz, S. Karayev, R. Girshick, T. Darrell, and K. Keutzer, “Densenet: Implementing efficient convnet descriptor pyramids,” *arXiv preprint arXiv:1404.1869*, 2014.

[29] K. He, E. Rozner *et al.*, “Presto: Edge-based load balancing for fast datacenter networks,” *ACM SIGCOMM Computer Communication Review*, 2015.

[30] M. Al-Fares, A. Loukissas, and A. Vahdat, “A scalable, commodity data center network architecture,” *ACM SIGCOMM computer communication review*, pp. 63–74, 2008.

[31] O. Russakovsky, J. Deng *et al.*, “ImageNet Large Scale Visual Recognition Challenge,” *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[32] V. Jalaparti, I. Bliznets *et al.*, “Dynamic pricing and traffic engineering for timely inter-datacenter transfers,” in *Proceedings of the 2016 ACM SIGCOMM Conference*, pp. 73–86.

[33] An instant virtual network on your laptop. [Online]. Available: http://mininet.org

[34] Behavioral model version 2 (bmv2). [Online]. Available: https://github.com/p4lang/behavioral-model

[35] A. Sapio, I. Abdelaziz, M. Canini, and P. Kalnis, “Daiet: a system for data aggregation inside the network,” in *Proceedings of the 2017 Symposium on Cloud Computing*.