GRID: Gradient Routing With In-Network Aggregation for Distributed Training

Jin Fang, Gongming Zhao, *Member, IEEE*, Hongli Xu, *Member, IEEE*, Changbo Wu, and Zhuolong Yu

***Abstract*— As the scale of distributed training increases, it brings huge communication overhead in clusters. Some works try to reduce the communication cost through gradient compression or communication scheduling. However, these methods either downgrade the training accuracy or do not reduce the total transmission amount. One promising approach, called in-network aggregation, is proposed to mitigate the bandwidth bottleneck in clusters by aggregating gradients in programmable hardware (*e.g.*, Intel Tofino switches). However, existing solutions mainly implement in-network aggregation through fixed (or default) routing paths, resulting in load imbalancing and long communication time. To deal with this issue, we propose GRID, the first-of-its-kind work on Gradient Routing with In-network Aggregation for Distributed Training. In the control plane, we present an efficient gradient routing algorithm based on randomized rounding and formally analyze the approximation performance. In the data plane, we realize in-network aggregation by carefully designing the logic of workers and programmable switches. We implement GRID and evaluate its performance on a small-scale testbed consisting of 3 Intel Tofino switches and 9 commodity servers. With a combination of testbed experiments and large-scale simulations, we show that GRID can reduce the communication time by 38.4%–60.1% and speed up distributed training by 17.4%–52.7% compared with state-of-the-art solutions.**

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

I. Introduction

As THE cornerstone of large-scale machine learning (ML) applications, distributed training (DT) is widely used in various fields (*e.g.*, computer vision [1], natural language processing [2] and recommender system [3]). In distributed training, compute nodes iteratively train large ML models for better performance (*e.g.*, higher classification accuracy).

Manuscript received 24 August 2022; revised 7 December 2022 and 12 January 2023; accepted 9 February 2023; approved by IEEE/ACM Transactions on Networking Editor V. Aggarwal. Date of publication 22 February 2023; date of current version 17 October 2023. This work was supported in part by the National Science Foundation of China (NSFC) under Grant 62132019 and Grant 62102392, in part by the Open Research Projects of Zhejiang Laboratory under Grant 2022QA0AB04, in part by the National Science Foundation of Jiangsu Province under Grant BK20210121, and in part by the Youth Innovation Promotion Association of CAS under Grant 2023481. *(Corresponding author: Gongming Zhao.)*

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

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

Digital Object Identifier 10.1109/TNET.2023.3244794

There are two kinds of compute nodes: workers and the parameter server (PS). In each iteration, workers perform gradient computation locally and send gradients to the PS. On receiving gradients from all the workers, the PS performs global aggregation and sends the results back to workers. As the scale of models and datasets grows, gradient aggregation requires massive communication resources, incurring performance bottleneck in practice [4], [5], [6], [7]. According to [7], for a DT task training DeepLight on 100Gbps links, 79% of the training time is occupied for communication.

To resolve this communication bottleneck, existing works often focus on communication scheduling [8], [9], [10], [11], [12] or gradient compression [13], [14], [15], [16]. Communication scheduling increases the overlap between computation and network transmission via fine-grained gradient transmission (*e.g.*, sub-models instead of the whole model). In this way, workers can reduce idle waiting time in each iteration and fully utilize network bandwidth. For example, ByteScheduler [10] accelerates distributed training by maximizing the overlap of gradient transmission and computation through Bayesian optimization. Though communication scheduling improves communication efficiency, it does not directly reduce the total transmission amount and may still encounter the communication bottleneck, especially on the PS side. Gradient compression can avoid the bandwidth bottleneck by reducing the volume of exchanged data. For example, the authors [6] exploit the sparsity of gradients to maximize effective bandwidth usage by sending only non-zero data blocks. But gradient compression faces the problem of degrading training accuracy.

Nowadays, programmable hardwares (*e.g.*, smart NICs [17], [18] and programmable switches [19]) provide the ability of computation. As a result, *in-network aggregation* has been proposed [5], [7], [20], [21] for mitigating the communication bottleneck of distributed training in clusters. Specifically, we can offload parts of gradient aggregation tasks into programmable hardware to reduce the amount of forwarded traffic. After a programmable device aggregates multiple gradients, only the aggregated gradient is transmitted in the network. For example, SwitchML [7] uses a P4-based programmable switch for aggregating the gradients of workers inside a rack to minimize the communication cost of a single rack. ATP [5] provides a protocol to support in-network aggregation in multi-tenant clouds. However, the above works mainly focus on efficiently realizing the aggregation operations in programmable switches, neglecting the question of *how to choose efficient gradient routing* (*i.e.*, where to perform in-network aggregation)? If we implement in-network aggregation with fixed (or default) routing paths, the traffic load distribution may be imbalanced, and the in-network aggregation capability cannot be fully utilized, leading to long communication time (see Section VI). Thus, *it is necessary to design proper gradient routing for in-network aggregation.*

However, performing gradient routing with in-network aggregation is non-trivial. On the one hand, distributed training tasks will face multi-dimensional resource constraints, such as switch processing capacity and link bandwidth. Moreover, the in-network aggregation will change the total amount of forwarded gradients, making existing routing methods [22], [23] inefficient. Therefore, designing an efficient gradient routing scheme with in-network aggregation is challenging. On the other hand, with limited in-memory size on programmable switches, it is expected to aggregate the synchronous gradients. However, due to network dynamics, gradient packets may arrive at programmable switches asynchronously, which should store a large number of intermediate results and exhaust the memory of programmable switches, decreasing the in-network aggregation throughput. Thus, it is necessary to design a rate synchronization mechanism to ensure that the gradients of multiple workers synchronously arrive at the switch for aggregation, which is also difficult. To handle these challenges, we design and implement GRID, which considers gradient routing with in-network aggregation in the context of clusters. The main contributions of this paper are as follows:

1) We propose GRID, a gradient routing framework for in-network aggregation, consisting of the control plane and the data plane, to mitigate the communication bottleneck and speed up the distributed training tasks.
2) We give a thorough control plane design and propose a randomized rounding based algorithm to maximize the gradient sending rate of workers with resource constraints. Moreover, we design and implement the data plane for workers and programmable switches, trying to synchronize the sending rate of workers.
3) We conduct a small-scale testbed based on Intel Tofino switches and large-scale simulation based on real-world network topologies [24], [25]. The experimental and simulation results show that, given the same number of training iterations, GRID can reduce the communication time by 38.4%-60.1% and speed up distributed training by 17.4%-52.7% compared with the state-of-the-art solutions.

The rest of this paper is organized as follows. In Section II, we summarize the state-of-art solutions to mitigate the communication bottleneck of distributed training. Section III presents a motivating example and overview of GRID. In Section IV, we illustrate the control plane design of GRID. The experimental and simulation results are presented in Section VI. We conclude this paper in Section VII.

II. Related Work

This section first introduces the situation of distributed training. Then, we illustrate how to speed up distributed training by communication scheduling. At last, we present how to mitigate the communication bottleneck through in-network aggregation.

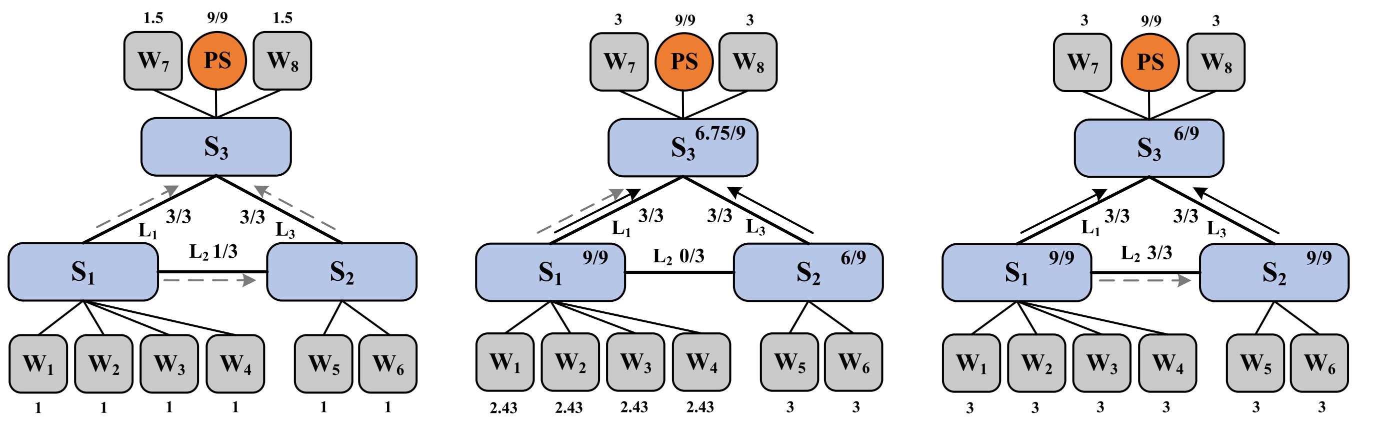
A. Distributed Training

A deep neural network (DNN) model consists of multiple network layers, each of which contains a large number of parameters. Training a DNN model requires hundreds of iterations over the dataset to achieve convergence [26]. In terms of the parallelism schemes, the distributed model training can be categorized into two main types: model parallelism and data parallelism [27]. This paper focuses on the data parallelism distributed model training, which splits the whole dataset into multiple compute nodes. In each iteration, each compute node independently trains the model on its partition of the dataset to generate the *gradient*. There are lots of algorithms for gradient calculation such as stochastic gradient descent (SGD) and its variants [28], [29], [30], [31], [32]. We take SGD as an example. Each compute node calculates gradient $g = \bigtriangledown f(w_t)$, where $\bigtriangledown$ denotes vector differential operator and $f(w_t)$ denotes the value of loss function related to model w_t in epoch t. Subsequently, compute nodes communicate with other nodes to update the global model parameters (*i.e.*, gradient aggregation) [33], [34], [35], [36], [37], [38]. This phase can be done asynchronously or synchronously. The former case mitigates the time of communicating at the cost of non-converging. The latter case can be acted as a synchronization barrier for convergence guarantees. For example, in synchronous SGD (SSGD) [34], the PS receives the gradients of workers and performs aggregation by calculating $\frac{1}{N}\sum_{n=1}^{N} g_t^n$, where N is the number of workers and g_t^n is the gradient of worker n in epoch t. In this paper, we consider the synchronization updates.

Parameter Server (PS) [39] and AllReduce [40] are two widely-adopted gradient aggregation schemes. In PS, there are two kinds of compute nodes: workers and parameter servers. Workers generate and push gradients to parameter servers. Afterward, parameter servers aggregate all the gradients and update the model parameters. At last, workers pull the updated results from parameter servers for the next training iteration. AllReduce uses collective communication operations to perform gradient aggregation. We take Ring-AllReduce [41], which is common in practice, as an example. In Ring-AllReduce, all compute nodes are workers and form a ring topology [42]. Each worker sends a partition of the gradient to its successor and receives another partition of the gradient from its predecessor. Recent studies have shown that the bottleneck in distributed model training is shifting from computing to communication [43]. To deal with this issue, communication scheduling and in-network aggregation are proposed to utilize the network bandwidth efficiently and reduce the traffic amount, respectively.

B. Communication Scheduling

Some works try to pipeline computing and communication phase of training, ranging from designing the high-performance traffic scheduler [8], [10], [12], [44] to optimizing collective communication operations [11], [45],



(a) Geryon (b) ATP (c) GRID

Fig. 1. A distributed training task containing 1 PS, 8 workers (*i.e.*, W_1-W_8), and 3 programmable switches (*i.e.*, S_1-S_3). The solid arrows represent the aggregated gradients, and the dotted arrows represent the non-aggregated ones. For simplicity, we omit the intra-rack transmission arrows. Let *load/capacity* represent the ratio of usage and capacity of links and programmable switches. The left plot shows the network workload of Geryon, which has a minimum gradient sending rate of 1 Gbps. The middle plot shows the network workload of ATP, which has a minimum gradient sending rate of 2.43 Gbps. The right plot shows the network workload of GRID, which has a minimum gradient sending rate of 3 Gbps.

[46], for accelerating the distributed training. For instance, the authors in [11] present BlueConnect, an efficient communication library optimized for GPU-based platforms, which decomposes a single all-reduce operation into numerous parallelizable operations to exploit the trade-off between communication time and bandwidth usage. Work [12] presents a traffic scheduler named Geryon, which determines the scheduling scheme for flows according to their priorities to maximize the utilization of bandwidth resources. However, these methods accelerate distributed training by overlapping the timing of computation and communication, while not directly reducing the amount of transmitted traffic. Therefore, these methods don't directly solve the problem of bandwidth exhaustion.

C. In-Network Aggregation

The idea of in-network aggregation begins at wireless networks [47], [48] and now attracts researchers to adopt in-network aggregation in clusters. Specifically, in-network aggregation offloads part of gradients aggregated in forwarding devices to reduce the amount of transferred data, alleviating the communication bottleneck.

Since in-network aggregation utilizes forwarding devices to aggregate gradients, it has the potential to co-exist with other methods performed in end hosts, *e.g.*, communication scheduling or gradient quantization [49], to further mitigate the communication bottleneck and accelerate distributed training.

There have been a lot of works implementing in-network aggregation in clusters with servers [43], [50], [51] or programmable switches [5], [7], [20], [21]. For example, NetAgg [50] uses dedicated servers connected with switches to perform in-network aggregation. However, server-based in-network aggregation incurs additional bandwidth costs and has limited scalability. With the rapid development of programmable switches (*e.g.*, P4-based [52], [53], FPGA-based [54]), performing in-network aggregation with programmable switches is becoming popular. For instance, SHARP [20] implements in-network aggregation based on a dedicated Mellanox's SiwtchIB-2 ASIC. iSwitch [21] tackles reinforcement learning and moves the gradient aggregation to FPGA-based programmable switches. PANAMA [55] designs an in-network hardware accelerator based on FPGA and presents a load-balancing protocol for in-network aggregation. As another option, P4-based programmable switches [56] attract a lot of attention. SwitchML [7] performs the in-network aggregation in a rack-scale network and offloads gradient aggregation of all workers to the top-of-rack (ToR) switches. ATP [5] considers a two-layer topology, and the gradients can be aggregated either on near-worker ToR switches or near-PS ToR switches. The authors in [57] consider the problem of how to place programmable switches in the network to minimize network overload.

However, these works mainly focus on efficiently realizing the aggregation operations in programmable switches, *ignoring the impact of gradient routing selection*. In fact, due to the constraint of switch processing capacity, the gradient routing selection is critical to the efficiency of in-network aggregation. Therefore, this paper design GRID to study the problem of gradient routing with in-network aggregation.

III. Motivation and Overview

This section first gives an example to illustrate the pros and cons of state-of-the-art solutions, which motivate our study. Then we present the overview and workflow of GRID.

A. A Motivating Example

Consider a distributed training task containing 1 PS and 8 workers. Each link has a bandwidth of 3 Gbps. Note that, in practice, the ingress bandwidth of the PS is often larger than that of workers to avoid the communication bottleneck, so we set the ingress bandwidth of the PS to 9 Gbps. The processing capacity of programmable switches is 9 Gbps.

Since the PS needs to wait for gradients of all workers to perform global aggregation, we take the minimum gradient sending rate as the critical metric, and the results are shown in Fig. 1. The circle represents the PS. The gray squares and blue rectangles represent workers W_1-W_8 and programmable switches S_1-S_3, respectively. We use *load/capacity* to denote the workload ratio and capacity for programmable switches and links. The solid arrows represent the aggregated gradients, and the dotted arrows represent the non-aggregated gradients sent by workers.

We first consider the Geryon scheme [12], which is a classical flow scheduling scheme in distributed training in Fig. 1(a). It schedules the gradients through different paths according to resource constraints to avoid network congestion. In this case, Geryon schedules the gradient of W_4 through the path W_4->S_1->S_2->S_3->PS to avoid congestion in link L_1. Accordingly, the gradients of workers W_1-W_3 are scheduled through the paths W_1->S_1->S_3->PS, W_2->S_1->S_3->PS and W_3->S_1->S_3->PS, respectively. Due to bandwidth constraints of links L_1 and L_3, workers W_1-W_6 will send the gradients with the minimum gradient sending rate of 1 Gbps.

We then consider a state-of-the-art method with in-network aggregation, named ATP [5]. In ATP, each worker chooses the nearest programmable switches for in-network aggregation (*i.e.*, S_1 aggregates W_1, W_2, W_3 and W_4. S_2 aggregates W_5 and W_6. S_3 aggregates W_7 and W_8). If the processing capacity of the programmable switch is exhausted, it will directly transfer the gradients to the PS. In this case, since the processing capacity of S_1 is 9 Gbps, W_1-W_4 can send gradients with the speed of 9/4=2.25 Gbps. Moreover, W_1-W_4 can send gradients with the additional speed of 0.75/4=0.18 Gbps to the PS, since link L_1 still has 3-2.25=0.75 Gbps available bandwidth. These gradients will be aggregated by S_3 with available processing capacity. As a result, the minimum gradient sending rate is 2.43 Gbps.

B. Our Intuition

From the above example, we observe that both solutions of accelerating distributed learning have advantages and disadvantages. Geryon routes the packets without in-network aggregation, therefore, the gradient sending rates of workers are limited by the ingress bandwidth of the PS. ATP adopts in-network aggregation, while routing the packets through the default paths to the PS. As a result, the gradient sending rates of workers W_1-W_4 are limited by programmable switch S_1. A question immediately following the above discussion is that *how to achieve efficient in-network aggregation through reasonable gradient routing under the resource constraints of both programmable switches and PS?*

In Fig. 1(b), we notice that, although S_1 can not aggregate gradients of W_1-W_4 with the speed of 3 Gbps, S_2 has available processing capacity for aggregating. Therefore, as shown in Fig. 1(c), we select S_2 to aggregate the gradients of W_4 and route these gradients through the path W_4->S_1->S_2. This way, all the workers can achieve the gradient sending rate of 3 Gbps. This scheme improves the minimum gradient sending rates by 200% and 23.5%, compared with Geryon and ATP, respectively. The reason for performance improvement is that, with heterogenous switch workloads, the fixed routing scheme of ATP will suffer the problem of load imbalancing. However,

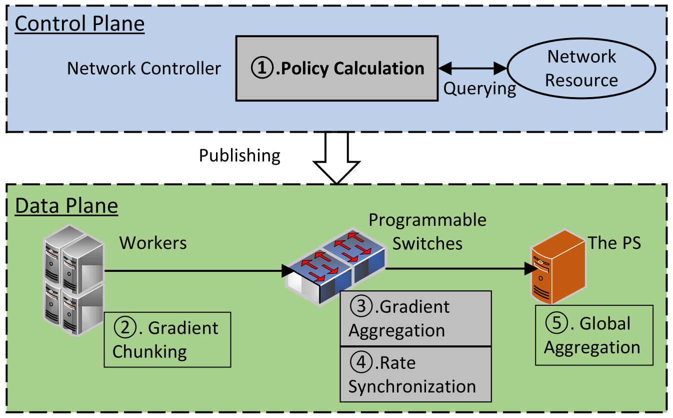


Fig. 2. System overview of GRID. GRID is composed of two parts. The control plane is responsible for determining the gradient routing policy. The data plane consists of workers, programmable switches and the PS, which is responsible for performing gradient routing with in-network aggregation.

our proposed scheme route workers' gradients to appropriate programmable switches according to their processing capacities. Motivated by this example, we design a gradient routing with an in-network aggregation framework called GRID. Note that, this example shows the performance gains for simplified distributed model training, further experimental results show that GRID can improve the gradient sending rate by 111.3% and 40.8%, compared with Geryon and ATP, respectively.

C. GRID Overview

As shown in Fig. 2, GRID consists of the control and data planes. The control plane leverages the collected network resource information to compute the gradient routing policy, *i.e.*, to which programmable switches (or the PS) each worker should send its gradient and the corresponding gradient sending rates. The data plane consists of workers, programmable switches and the PS. Specifically, workers are responsible for gradient chunking. Programmable switches perform gradient aggregation and rate synchronization to realize the gradient routing with in-network aggregation. At last, the PS performs global aggregation.

The core of GRID is to determine the gradient routing policy, which will be introduced in Section IV. For the data plane, we can implement gradient routing and aggregation based on existing solutions [5]. Therefore, we present the workflow in Section III-D and illustrate the detailed design of gradient aggregation and rate synchronization in Section V.

D. Workflow of GRID

Fig. 2 briefly illustrates the GRID workflow, which mainly consists of 5 steps.

1) Policy calculation: The network controller calculates the gradient routing policy and publishes the policy to workers and programmable switches. Note that, the gradient routing policy is published only once, and the data plane will iteratively execute the following 4 steps in the DT task.
2) Gradient chunking: Since the switch memory size is usually smaller than the gradient size, existing works [5],

TABLE I
TABLE OF NOTATIONS

Notations	Semantics
α	the PS
N	the set of workers
S	the set of programmable switches
B_α	the ingress bandwidth of the PS α
C_s	the processing capacity of programmable switch s
T	the maximum sending rate
x_n^α	whether the gradient of the worker n is aggregated by the PS or not
x_n^s	whether the gradient of the worker n is aggregated by programmable switch s or not
r_n^α	the sending rates of worker n to the PS
r_n^s	the sending rates of worker n to programmable switch s
y_s	the sending rate of programmable switch s to the PS
λ	the minimal sending rates among workers

[7] perform aggregation in switch with the granularity of gradient fragment. Specifically, each worker splits its gradient into a set of gradient fragments, each of which can be identified by the tuple of <*node id*, *fragment id*>. The *node id* indicates which programmable switch or the PS will aggregate the gradient fragment. The *fragment id* is the index of the fragment. Each switch organizes its memory as an array of memory units, each of which can aggregate one fragment at a time.

3) Gradient aggregation: Each programmable switch maintains an identity id. On receiving a gradient fragment, the programmable switch compares the fragment's *node id* with its id. If they match, the programmable switch will perform gradient aggregation. Otherwise, the programmable switch will directly forward the fragment according to the flow table and perform rate synchronization.
4) Rate synchronization: Once a hash collision happens, the programmable switch will send a control packet to inform the corresponding worker adjusting the size of sending window.
5) Global aggregation: The PS collects all gradient fragments (aggregated by programmable switches and directly sent from workers) and performs aggregation.

IV. GRID Control Plane Design

Determining gradient routing policy is the key step in the control plane of GRID. To achieve efficient in-network aggregation, we first formulate the problem of Gradient Routing with In-network Aggregation (GRIA). Then we propose a randomized rounding based approximation algorithm named R-GRIA. At last, we analyze the approximation performance of R-GRIA.

A. Network Model

Parameter server architecture. A parameter server architecture consists of the parameter server (PS) α and a set of workers $N = \{n_1, n_2, \ldots, n_{|N|}\}$. Workers compute the gradients locally and send these gradients to the PS with the rate of r_n^α. The ingress bandwidth of the PS α is denoted by B_α. Note that, our algorithm can be easily extended to architectures with multiple PSs, since the partitions of each PS are independent.

Programmable network. We consider a cluster (*e.g.*, datacenter) containing four elements: a compute node set, a programmable switch set, a link set and a network controller.

1) The workers and the PS are hosted on the compute nodes for gradients calculation and global aggregation.
2) The programmable switches are responsible for forwarding and gradient aggregation. Let $S = \{s_1, s_2, \ldots, s_{|S|}\}$ denote the programmable switch set. Each programmable switch s has a limited processing capacity C_s.
3) The compute nodes and the programmable switches are connected via a set of links. Since the network topology is stable in datacenters and elements are connected with high bandwidth links, we assume the link bandwidth is sufficient.
4) The network controller can manage the whole network, *e.g.*, routing the gradients of the workers.

B. Problem Formulation

This section gives the problem formulation of the Gradient Routing with In-network Aggregation (GRIA) problem in clusters. Supposing that the workers forward the gradients to the PS, some of the gradients are forwarded and aggregated in-network by programmable switches. This routing scheme can be split into two phases: 1) workers' gradients are forwarded to programmable and 2) programmable switches perform aggregation and forward the aggregated gradients to the PS. In the following, we use aggregation nodes to represent programmable switches and the PS, since they all have gradient aggregation capabilities. For each worker $n \in N$, we should determine the aggregation node. We use $x_n^s \in \{0,1\}$ to represent the gradient of the worker n is aggregated by the programmable switch s or not, and $x_n^\alpha \in \{0,1\}$ to denote whether the gradient is aggregated by the PS, or not. We use r_n^α and r_n^s to denote the sending rates of worker n to the PS α and the programmable switch s, respectively. Once a programmable switch s performs aggregation, the result should be sent to the PS α. Let y_s represent the sending rate of the programmable switch s to the PS, if it performs the in-network aggregation. We illustrate the following constraints of the GRIA problem:

1) *Aggregation Constraint*: Considering the number of programmable switches in the cluster is limited, routing gradients through multiple programmable switches will cause longer routing paths and higher bandwidth consumption. Similar to [5], we assume that each gradient will be aggregated in-network once at most to balance the complexity and the performance of the GRIA problem, we have $\sum_{s \in S \cup \{\alpha\}} x_n^s = 1, \forall n \in N$.
2) *Sending Rate Constraint*: For each worker $n \in N$, its sending rate to the aggregation node s can't exceed the

maximum sending rate T (*i.e.*, the ingress bandwidth of the PS), that is $r_n^s \le x_n^s \cdot T, \forall n \in N, s \in S \cup \{\alpha\}$.

3) *Aggregation Node Constraint*: For each worker $n \in N$, its sending rate can't exceed that of corresponding programmable switches, if some workers choose it as the aggregation node, which means $r_n^s \le y_s, \forall n \in N, s \in S$.
4) *Processing Capacity Constraint*: Each programmable switch can aggregate gradient with a limited processing rate, which is $\sum_{n\in N} r_n^s \le C_s, \forall s \in S$.
5) *Bandwidth Constraint*: The forwarding rate can't exceed the ingress bandwidth of the PS α. There are two kinds of flows. For each worker $n \in N$, if its aggregation node is the PS, its flow consumes the ingress bandwidth of the PS by r_n^α. Otherwise, worker n's gradient is aggregated by the programmable switch s, and only one aggregated flow will be sent to the PS. Therefore, it only consumes the ingress bandwidth of the PS by y_s. The bandwidth constraint can be represented as $\sum_{n\in N} r_n^\alpha + \sum_{s\in S} y_s \le B_\alpha$.

With these constraints, the problem can be formulated as follows:

$$\max \ \lambda$$
$$S.t. \begin{cases} \sum_{s\in S\cup\{\alpha\}} x_n^s = 1, & \forall n \in N \\ r_n^s \le x_n^s \cdot T, & \forall n \in N, s \in S \cup \{\alpha\} \\ r_n^s \le y_s, & \forall n \in N, s \in S \\ \lambda \le \sum_{s\in S\cup\{\alpha\}} r_n^s, & \forall n \in N \\ \sum_{n\in N} r_n^s \le C_s, & \forall s \in S \\ \sum_{n\in N} r_n^\alpha + \sum_{s\in S} y_s \le B_\alpha & \\ x_n^s \in \{0,1\}, & \forall n \in N, s \in S \cup \{\alpha\} \\ r_n^s \ge 0, & \forall n \in N, s \in S \cup \{\alpha\} \\ y_s \ge 0, & \forall s \in S \end{cases} \tag{1}$$

The first set of equations denotes the aggregation constraint. The second set of inequalities represents the sending rate constraint. The third set of inequalities means the aggregation node constraint. We define λ as the minimum sending rate among the workers and the fourth set of inequalities calculates the λ. The fifth set of inequalities denotes the processing capacity constraint. The sixth inequality represents the bandwidth constraint. Our goal is to maximize the minimum sending rate of workers.

C. Algorithm Design

In this section, we propose a randomized rounding based algorithm for the GRIA problem, called R-GRIA. Specifically, R-GRIA routes the gradients via two major steps: 1) Relaxing the constraints of GRIA for computing the optimal solutions; 2) Determining the in-network aggregation scheme, including calculating the aggregation nodes and gradient sending rates of each worker.

In the first step, we relax Eq. (1) by replacing $x_n^s \in \{0,1\}$ with $x_n^s \in [0,1]$. Then, we can solve it with a linear program solver (*e.g.*, PULP [58]) and the optimal solutions are denoted as $\{\widetilde{x}_n^s, \widetilde{r}_n^s, \widetilde{y}_s\}$. In the second step, we select the aggregation node based on the optimal solutions. Specifically, for each worker n, the algorithm chooses the aggregation node $s \in S \cup \{\alpha\}$ with the probability $\frac{\widetilde{r}_n^s}{\sum_{s\in S\cup\{\alpha\}} \widetilde{r}_n^s}$. The aggregation node of worker n is denoted as s_n. After all the workers have selected the aggregation nodes, we define $S' = \{s | s \in S, \sum_{n\in N} \widehat{x}_n^s > 0\}$ to denote the programmable switches, which perform in-network aggregation. We set the gradient sending rate of worker $n \in N$ to $r_n^{s_n} = \sum_{n\in S\cup\{\alpha\}} \widetilde{r}_n^s$ and that of programmable switch $s \in S'$ to$\widehat{y}_s = \max\{r_n^{s_n}, n \in N\}$, respectively. As a result, the value of λ is set to $\lambda = \min\{r_n^{s_n}, n \in N\}$. The algorithm is summarized in Alg. 1.

Algorithm 1 R-GRIA: Randomized Rounding Algorithm for GRIA

```
1:  Step 1: Solving the Relaxed Problem
2:  Construct a linear programming $LP$ by replacing with $x_n^s \in [0,1]$.
3:  Derive the optimal solutions $\{\widetilde{x}_n^s, \widetilde{r}_n^s, \widetilde{y}_s\}$.
4:  Step 2: Determining the In-network Aggregation Scheme
5:  for each worker $n \in N$ do
6:      Choose the programmable switch or PS $s \in S\cup\{\alpha\}$ as the aggregation node with the probability $\frac{\widetilde{r}_n^s}{\sum_{s\in S\cup\{\alpha\}} \widetilde{r}_n^s}$ and set $\widehat{x}_n^s = 1$.
7:      Let $s_n$ denote the aggregation node of worker $n$.
8:      Set the gradient sending rate to $r_n^{s_n} = \sum_{s\in S\cup\{\alpha\}} \widetilde{r}_n^s$.
9:  end for
10: Define $S' = \{s | s \in S, \sum_{n\in N} \widehat{x}_n^s > 0\}$.
11: for each programmable switch $s \in S'$ do
12:     Set the gradient sending rate to $\widehat{y}_s = \max\{r_n^{s_n}, n \in N\}$
13: end for
14: Set $\lambda = \min\{r_n^{s_n}, n \in N\}$.
```

D. Performance Analysis

Theorem 1: R-GRIA can guarantee that each worker selects one aggregation node. (*i.e.*, R-GRIA guarantees the *Aggregation Constraint.*)

Proof: In line 6 of the Alg. 1, R-GRIA only chooses one place with the probability $\frac{\widetilde{r}_n^s}{\sum_{n\in S\cup\{\alpha\}} \widetilde{r}_n^s}$ as the aggregation node. Thus, the gradients are aggregated on one aggregation node. □

Theorem 2: R-GRIA guarantees that for each worker, its sending rate won't exceed the maximum sending rate T.

Proof: According to Eq. (1), we can know that $\widetilde{r}_n^s \le \widetilde{x}_n^s \cdot T, \forall n \in N, s \in S \cup \{\alpha\}$ and $\sum_{s\in S\cup\{\alpha\}} \widetilde{x}_n^s \le 1, \forall n \in N$. Combining these two inequalities, we have:

$$r_n^{s_n} = \sum_{n\in S\cup\{\alpha\}} \widetilde{r}_n^s \le \sum_{n\in S\cup\{\alpha\}} \widetilde{x}_n^s \cdot T \le T \tag{2}$$

As a result, we guarantee the *Sending Rate Constraint.* □

Theorem 3: R-GRIA guarantees that for each worker, its sending rate won't exceed that of its aggregation node.

Proof: In line 11 of Alg. 1, R-GRIA sets $\widehat{y}_s = \max\{r_n^{s_n}, n \in N\}$ to ensure the gradient sending rate of programmable switches won't lower than that of workers. □

Lemma 4: **Chernoff Bound**: 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, $\Pr\left[\sum_{i=1}^{n} y_i \geq (1+\varrho)\tau\right] \leq e^{\frac{-\varrho^2\tau}{2+\varrho}}$, where ϱ is an arbitrary positive value.

Theorem 5: R-GRIA will not exceed the *Processing Capacity Constraint* by an approximation factor of $O(\log|S|)$. Under the proper assumption, the bound can all be tightened to 2.

Proof: We first prove that for each worker $n \in N$ and aggregation node $s \in S \cup \{\alpha\}$, we have $\mathbb{E}\left[\widehat{r}_n^s\right] = \widetilde{r}_n^s$. Since we choose the aggregation node s_n for worker n with the probability of $\frac{\widetilde{r}_n^s}{\sum_{n \in S\cup\{\alpha\}} \widetilde{r}_n^s}$, and set the gradient sending rate of worker n as $\widehat{r}_n^s = \sum_{n \in S \cup\{\alpha\}} \widetilde{r}_n^s$. The expected value of $\widehat{r}_n^s$ is:

$$\mathbb{E}\left[\widehat{r}_n^s\right] = \frac{\widetilde{r}_n^s}{\sum_{n \in S\cup\{\alpha\}} \widetilde{r}_n^s} \cdot \sum_{n \in S\cup\{\alpha\}} \widetilde{r}_n^s = \widetilde{r}_n^s \tag{3}$$

Then we define $\delta_s = \sum\limits_{n \in N} \widehat{r}_n^s$ as the processing throughput of the programmable switch s. Since each worker n selects the programmable switch s as the aggregation node independently, we have $\mathbb{E}\left[\delta_s\right] = \sum_{n \in N} \widetilde{r}_n^s$. By the definition of δ_s, we can get the expected computing workload of each programmable switch $s \in S$:

$$\mathbb{E}\left[\delta_s\right] = \textstyle\sum_{n \in N} \widetilde{r}_n^s \leq C_s \tag{4}$$

Let $C_{\min}$ denote the minimum processing capacity among the programmable switches. We then define a constant value $\nu = \frac{C_{\min}}{N \cdot T}$ to normalize the expected computing workload. Combining Eq. (4) and the definition of ν, we have:

$$\begin{cases} \frac{\delta_s \cdot \nu}{C_s} \in [0,1] \\ \mathbb{E}\left[\frac{\delta_s \cdot \nu}{C_s}\right] \leq \nu \end{cases} \tag{5}$$

By applying Lemma 4, we have:

$$\begin{aligned} &\Pr\left[\frac{\delta_s \cdot \nu}{C_s} \geq (1+\varrho)\cdot\nu\right] \leq e^{\frac{-\varrho^2\nu}{2+\varrho}} \\ \Rightarrow &\Pr\left[\frac{\delta_s}{C_s} \geq (1+\varrho)\right] \leq e^{\frac{-\varrho^2\nu}{2+\varrho}} \end{aligned} \tag{6}$$

where ϱ is an arbitrary positive value.

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

$$\Pr\left[\frac{\delta_s}{C_s} \geq (1+\varrho)\right] \leq e^{\frac{-\varrho^2\nu}{2+\varrho}} \leq \frac{1}{|S|} \tag{7}$$

which means that the upper bound approaches quickly to zero as the network grows. By solving Eq. (7), we have:

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

In practice, the processing capacity of a programmable switch can achieve up to 3.2Tbps [52], a PS architecture contains 8-36 workers in general [59], and we set $|N| = 36$ here. In the current datacenter, the ingress bandwidth of the PS can achieve up to 100 Gbps [60]. Under this setting, $\nu = \frac{320000}{35\cdot 100} \approx 91.43$. We assume the number of programmable switches in a datacenter is $|S| = 50$, so $3 \cdot \log|S| \approx 5.09$. Combining these assumptions, we can obtain that $\nu \geq 3 \cdot \log|S|$. As a result, we have:

$$\begin{aligned} &\varrho \geq \frac{\log|S| + \sqrt{\log^2|S| + 8\nu\log|S|}}{2\nu} \\ \Rightarrow\ &\varrho \geq \frac{\log|S| + \sqrt{(2\nu - \log|S|)^2 + 12\nu\log|S| - 4\nu^2}}{2\nu} \\ \Rightarrow\ &\varrho \geq \frac{\log|S| + \sqrt{(2\nu - \log|S|)^2}}{2\nu} \Rightarrow \varrho \geq 1 \end{aligned} \tag{9}$$

Thus, the approximate factor of the *Processing Capacity Constraint* is $(\varrho + 1) = \frac{\log|S|}{\nu} + 3 = O(\log|S|)$. Under the proper assumption (*i.e.*, $\nu \geq 3 \cdot \log|S|$), the bound can be tightened to $\varrho + 1 = 2$. □

Theorem 6: R-GRIA will not exceed the *Bandwidth Constraint* by an approximate factor of $O(\log|N \cdot S|)$. Under the proper assumption, the bound can be tightened to 4.

Proof: According to the bandwidth constraint, for the ingress link of the PS α, there are two kinds of flows going through: the flows from the workers and the flows from the programmable switches. We define $B_{\alpha,1} = \sum\limits_{n \in N} \widehat{r}_n^\alpha$ as the bandwidth usage of the first set of flows and $B_{\alpha,2} = \sum\limits_{s \in S} \widehat{y}_s$ as the bandwidth usage of the second set of flows, respectively. The bandwidth usage of the PS ingress link can be represented as $B_{\alpha,1} + B_{\alpha,2} \leq B_\alpha$.

We first consider the bandwidth usage of the first flow set, which can be calculated as $\sum\limits_{n \in N} \widehat{r}_n^\alpha = B_{\alpha,1}$. Let variable $\sigma_n^\alpha = \widehat{r}_n^\alpha$ represent the gradient sending rate of worker n. Since the worker n can send gradient to the PS, if and only if it selects the PS as the aggregation node, we have:

$$\mathbb{E}\left[\sigma_n^\alpha\right] = \sum_{s \in S\cup\{\alpha\}} \widetilde{r}_n^s \cdot \frac{\widetilde{r}_n^\alpha}{\sum\limits_{s \in S\cup\{\alpha\}} \widetilde{r}_n^s} = \widetilde{r}_n^\alpha \tag{10}$$

Thus, the expected bandwidth consumption of the first flow set is:

$$\begin{aligned} \mathbb{E}\left[\sum_{n \in N} \sigma_n^\alpha\right] &= \sum_{n \in N} \mathbb{E}\left[\sigma_n^\alpha\right] \\ &= \sum_{n \in N} \widetilde{r}_n^\alpha \end{aligned} \tag{11}$$

Using similar methods as in Theorem 5, we can prove that the bandwidth usage of the first set of flows won't be violated by an approximation factor of $O(\log|N|)$, which means

$$\sum_{n \in N} \widehat{r}_n^\alpha \leq O(\log|N|) \cdot B_{\alpha,1} \tag{12}$$

We next consider the bandwidth usage of the second flow set, which can be calculated as $\sum_{s \in S} \widehat{y}_s = B_{\alpha,2}$. By observing the optimal results of LP in Alg. 1, we can assume that all the workers with the same aggregation node s_n have the same sending rates. As a result, we have $\widetilde{y}_{s_n} = \widetilde{r}_n^{s_n}, \forall n \in N$. We use n' to denote the worker with the largest sending rate to the programmable switch s. The expected bandwidth consume by the programmable switch s is

$$\mathbb{E}\left[\widehat{y}_s\right] = \mathbb{E}\left[\widehat{r}^s_{n'}\right] = \widetilde{r}^s_{n'} = \widetilde{y}_s \tag{13}$$

According to Eq. (13), we can calculate the expected bandwidth consume of the second set of flows as:

$$\begin{aligned}\mathbb{E}\left[\sum_{s\in S}\widehat{y}_s\right] &= \sum_{s\in S}\mathbb{E}\left[\widehat{y}_s\right]\\ &= \sum_{n\in N}\widetilde{y}_s \end{aligned}\tag{14}$$

Therefore, we can prove that the bandwidth usage of the second set of flows won't be violated by an approximation of $O(\log|S|)$:

$$\sum_{s\in S}\widehat{y}_s \leq O(\log|S|)\cdot B_{\alpha,2} \tag{15}$$

Combining the Eq. (12) and the Eq. (15), the total bandwidth violation factor won't exceed:

$$\begin{aligned}&\frac{O(\log|N|)\cdot B_{\alpha,1} + O(\log|S|)\cdot B_{\alpha,2}}{B_\alpha}\\ &\leq O(\log|N| + \log|S|) = O(\log|N\cdot S|)\end{aligned}\tag{16}$$

Besides, similar to Theorem 5, we can prove that under the assumption of $B_{\alpha,1} \geq 3\cdot\log|N|$ and $B_{\alpha,2} \geq 3\cdot\log|S|$, the approximation factor of the *Bandwidth Constraint* can be tightened to 4. □

Theorem 7: After rounding, the minimum sending rate of workers will equal the value in the relaxed LP. (*i.e.*, we guarantee that λ is the optimal result.)

Proof: In line 12 of the Alg. 1, we set $\lambda = \min\{r^{s_n}_n, n \in N\}$, where $r^{s_n}_n = \sum_{s\in S\cup\{\alpha\}}\widetilde{r}^s_n$. We define $\widetilde{\lambda}$ as the optimal result in the LP. By observing the LP in Alg. 1, we have

$$\begin{aligned}\widetilde{\lambda} &= \min\left\{\sum_{s\in S\cup\{\alpha\}}\widetilde{r}^s_n, n\in N\right\}\\ &= \min\{r^{s_n}_n, n\in N\} = \lambda\end{aligned}\tag{17}$$

Eq. (17) means that the R-GRIA algorithm can guarantee that the value of λ will equal that in the fractional solution after randomized rounding. □

V. GRID Data Plane Design

Although the GRID controller determines the gradient routing policy, there are still two issues when implementing in-network aggregation for each programmable switch. The first issue is how programmable switches determine which gradients are responsible for aggregating when receiving gradients from different workers. The second issue is that when gradient fragments arrive at switches asynchronously, how does the programmable switch synchronize worker sending rates so that in-network aggregation throughput remains high? To address both issues, this section describes the detailed design of gradient aggregation and rate synchronization.

A. Gradient Aggregation

When a programmable switch needs to aggregate a gradient fragment, it hashes the fragment into a memory unit according to the *fragment id*. Specifically, supposing that the number of memory units is M, the switch will compute Hash(<fragment id>)$\%M$ to allocate the gradient fragment. The switch records the *fragment id* and the aggregation count. If the corresponding memory unit is empty, it will store the value of the gradient fragment, and the switch sets the aggregation count to 1. Otherwise, it compares the *fragment id* with its record. If the *fragment id* is not matched, we say a collision happens. Since the corresponding memory unit is aggregating another gradient fragment, the switch drops the incoming fragment and performs rate synchronization. If the *fragment id* is matched, the switch accumulates the value of the incoming fragment into the stored values and increments the aggregation count. Once the programmable switch finishes the gradient fragment's aggregation (*i.e.*, the number of aggregation equals the aggregation number of the routing policy), it will send this fragment to the PS, releasing the corresponding memory unit for aggregating the following gradient fragments.

B. Rate Synchronization

Once a collision happens, the programmable switch will send a control packet to inform the corresponding worker to adjust the gradient sending rate. The reason for collision is that the gradient fragment with the larger *fragment id* arrives at the programmable switch, while the corresponding memory unit doesn't complete the aggregation of the gradient fragment with the smaller *fragment id*. Therefore, the programmable switch turns down the gradient sending rate of the worker with the larger *fragment id* to avoid the asynchronous arrival of gradient fragments. Like TCP, the programmable switches use ECN marks as the signal of control packets. Workers use sending window size to control gradient sending rates. Each worker applies additive increase multiplicative decrease to adjust its window size in response to aggregated gradients and control packets [5]. At the begining, workers maintain the same window size. When workers receive the aggregated packets from the PS, they increase the window size by one MTU until it reaches a threshold. On receiving a control packet, the worker halves the window size and updates the threshold of the updated window to turn down the gradient sending rates. The worker will resend the gradient packet with the same *fragment id* as the control packet to continue the aggregation.

VI. Performance Evaluation

In this section, we compare GRID with state-of-the-art solutions. We first give the metrics and benchmarks for performance comparison (Section VI-A). Then, we construct a small-scale testbed with Wedge100BF-32x programmable switches [52] to test the efficiency of GRID (Section VI-B). Finally, to complement the small-scale testbed experiments, we perform simulations to show the *theoretical performance* of GRID in large-scale scenarios (Section VI-C).

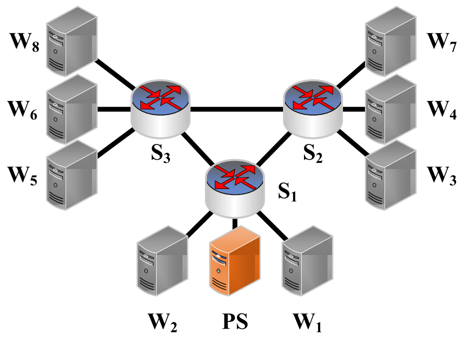


Fig. 3. Topology of the testbed consisting of 1 PS, 8 workers (W_1-W_8) and 3 programmable switches (S_1-S_3). All the components are connected via 100 Gbps links.

A. Performance Metrics and Benchmarks

Metrics. We adopt the following performance metrics for performance comparison: (1) the training throughput; (2) the gradient sending rate; (3) the per-iteration time; (4) the communication time per iteration; (5) the test accuracy; (6) the aggregation rate of programmable switches; (7) the aggregation rate of the PS.

During a testbed run, we measure the number of processed samples (*e.g.*, images) per second as the *training throughput*. We use iftop [61] to monitor the egress bandwidth usage of each worker as the *gradient sending rate*. Moreover, we record the time between two consecutive iterations as the *per-iteration time*. In each iteration, we measure the duration starting from the time a worker starts sending a gradient till the time that worker receives the aggregated gradient as the *comunication time per iteration*. Besides, we measure the proportion between the amount of the data correctly predicted by the model to that of all data in the test set as the *test accuracy*.

During a simulation run, we measure the total amount of gradients aggregated by switches per second as the *aggregation rate of programmable switches*. We measure the ingress bandwidth load of the PS, as the *aggregation rate of the PS*.

Benchmarks. We compare GRID with three benchmarks. The first benchmark is a communication scheduling scheme without considering in-network aggregation, called Geryon [12]. Geryon selects the shortest path to the PS for each gradient under the resource constraints. The second one is ATP [5], which performs in-network aggregation at multiple racks of programmable switches. Each worker sends the gradient to the PS via pre-defined routing paths, where the gradient is aggregated in the first encountered aggregation node with available processing capacity. The last one is an in-network aggregation framework called SwitchML [7], which minimizes the communication overheads at each rack. For fair evaluation, we further accelerate the training of SwitchML by sending the aggregated gradients of the programmable switch to the PS for global aggregation.

B. Testbed Evaluation

Settings. We use 9 servers and 3 Wedge100BF-32x programmable switches [52] to build the testbed. The topology of the testbed is shown in Fig. 3. Specifically, each server

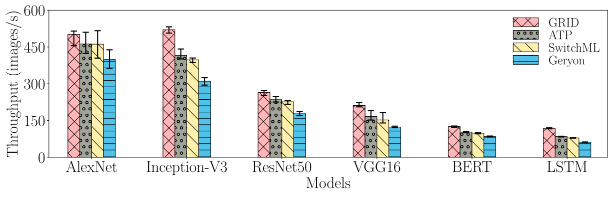


Fig. 4. Training throughput vs. models.

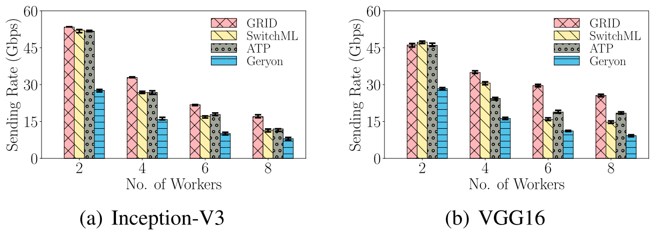


Fig. 5. Gradient sending rate vs. no. of workers.

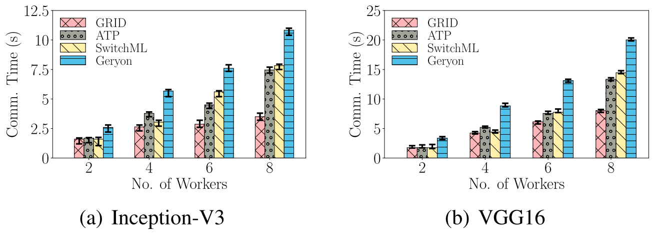
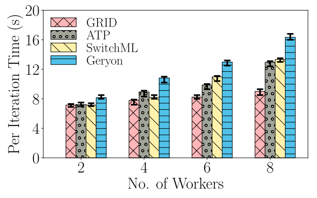


Fig. 6. Communication time vs. no. of workers.

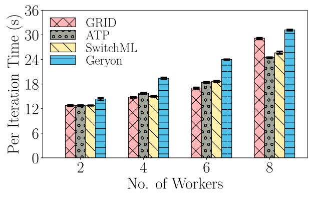
has one NVIDIA GeForce RTX 3090, a 22-core Intel Xeon 6152 processor and a Mellanox ConnectX-6 100G dual-port NIC. All the servers run Ubuntu 18.04 with CUDA 11.6. The NIC driver of all servers is Mellanox driver OFED 5.5-1.0.3.2. All programmable switches feature Intel Tofino chip with Software Development Environment (SDE) 9.7.0 [62]. Moreover, these servers and programmable switches are connected by 100 Gbps links as shown in Fig. 3.

Similar to [5], each server runs PyTorch [63] to perform distributed training tasks. For the gradient routing policies, we pre-calculate the routing scheme of our algorithm with PuLP [58] in the PS. After that, the PS publishes the routing policies to programmable switches and workers. Specifically, for each switch, the PS can connect it with the Secure Shell (SSH) protocol and publishes a unique id and a forwarding table using the Barefoot Runtime Interface (BRI). For each worker, the PS publishes the id of its aggregation node using the DistributedDataParallel module provided by Pytorch. We implement in-network aggregation in programmable switches by writing the P4-16 program for the Tofino Native Architecture (TNA). The programmable switches can aggregate 64 elements per gradient packet using one switch pipeline. We implement routing and aggregation logic in multiple stages of the ingress pipeline and packet control logic in the egress pipeline.

Overall performance comparision. In this set of evaluations, we run several popular models: AlexNet [64], Inception-V3 [65], ResNet50 [1], VGG16 [66], BERT [2] and LSTM [67], to evaluate the training throughput performance.

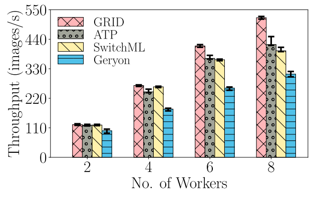


(a) Inception-V3

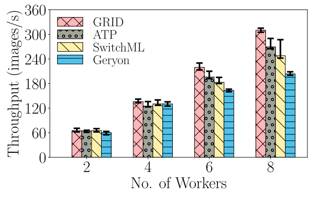


(b) VGG16

Fig. 7. Per-iteration time vs. no. of workers.



(a) Inception-V3



(b) VGG16

Fig. 8. Training throughput vs. no. of workers.

We train AlexNet, Inception-V3, ResNet50 and VGG16 on Cifar-100 dataset [68] and BERT and LSTM on wikitext-2 [69] dataset. The batch size is set as 64 for all models. As shown in Fig. 3, the network topology consists of 8 workers and 1 PS, and overall performance results are shown in Fig. 4. We can see that GRID obtains the highest training throughput among the four algorithms. For example, GRID achieves the throughput of 267 images/s on average when training ResNet50, while ATP, SwitchML and Geryon obtain the throughputs of 239 images/s, 228 images/s and 181 images/s, respectively. When the model is VGG16, GRID increases the throughput by 27.2%, 38.5% and 71.3% on average, compared with ATP, SwitchML and Geryon, respectively. The reason is that GRID decreases the communication time by selecting optimal gradient routing policy to perform efficient in-network aggregation. To save space, we only conduct a detailed performance comparison of all solutions with Inception-V3 and VGG16.

Performance comparision on per iteration training. In this set of evaluations, we estimate the performance of per iteration training. The network initially contains 2 workers (W_1, W_2), 1 programmable switch (S_1) and the PS. Then we add 2 workers (W_3, W_4) and 1 programmable switch (S_2) to the network. Similarly, we then add W_5, W_6 and S_3 to the network. The final topology contains 8 workers, 3 programmable switches and the PS (*i.e.*, Fig. 3). The evaluation results are shown in Figs. 5-8. From Fig. 5, we can see that GRID can always achieve the highest gradient sending rate as the number of workers increases. For example, given 8 workers in Fig. 5(a), the communication throughput of GRID, ATP, SwitchML and Geryon are 16.9 Gbps, 12.0 Gbps, 11.4 Gbps and 8 Gbps, respectively. GRID can increase the gradient sending rates by 40.8%, 48.2% and 111.3%, compared with ATP, SwitchML and Geryon, respectively. The reason is that GRID can select efficient routing paths and aggregation nodes by leveraging the proposed R-GRIA algorithm. Fig. 6 shows that GRID always has the least communication time in each iteration. For example, when the number of workers is 8 in Fig. 6(b), GRID decreases the communication time by 38.4%, 41.2% and 60.1%, compared with ATP, SwitchML and Geryon, respectively. The reason is that GRID has the highest gradient sending rate (as described in Fig. 5), thus reducing the communication time. Fig. 7 shows the per-iteration time with different numbers of workers. Note that per-iteration time consists of the local training and communication time. Our method doesn't optimize the local training time but can co-exist with solutions decreasing local training time if needed. We can see that, as the number of workers increases, the per-iteration time increases too, while GRID always obtains the least per-iteration time. Given 8 workers in Fig. 7(b), the per-iteration times of GRID, ATP, SwitchML and Geryon are 19.2s, 24.5s, 25.6s and 31.2s, respectively. Thus, we can conclude that by decreasing the communication time, GRID reduces the per-iteration time by 21.6%, 25% and 38.5%, compared with ATP, SwitchML and Geryon, respectively. Fig. 8 shows that as the number of workers increases, GRID can always obtain the highest training throughput. For example, given 8 workers in Fig. 8(b), the training throughput of GRID is 17.4%, 25.5% and 52.7% higher than that of ATP, SwitchML and Geryon, respectively. The reason is that we have designed the R-GRIA algorithm to select a better gradient routing scheme with in-network aggregation, thereby increasing the gradient sending rate.

Performance comparison on end-to-end training. In this set of evaluations, we run two end-to-end distributed training tasks to evaluate the performance of training time and accuracy. Specifically, we set the number of training iterations of Inception-V3 and VGG16 to 200 and 500, respectively. From Figs. 9-10, we can see that GRID always takes the least time to complete the same number of iterations compared with other alternatives. Fig. 9 shows that, GRID takes the least time to complete the distributed training task. Fig. 10 further indicates that GRID can obtain the specified test accuracy with the least time. For instance, in Fig. 10, when the mode is Inception-V3, GRID takes 1830s to finish all the training iterations, while ATP, SwitchML and Geryon take 2592s, 2666s and 3268s to complete. Besides, GRID first achieves an accuracy of 0.7925 in 968s, while that time of ATP, SwitchML and Geryon are 1568.16s, 1612.93s and 1977.14s, respectively. It means that GRID can reach the target accuracy 1.61×, 1.66× and 2.04× faster than ATP, SwitchML and Geryon. The results show that proper gradient routing with in-network aggregation can significantly speed up the distributed model training.

Summary. From the above testbed evaluation results, we can draw some conclusions. First, Fig. 4 shows that GRID can improve the training throughput by about 30% on average compared with other alternatives for distributed model training. Second, in Figs. 5-8, we can see that GRID reduces per-iteration time by about 25% on average compared with ATP, SwitchML and Geryon. At last, from Figs. 9-10, we believe GRID can achieve similar test accuracy with other alternatives, which takes more time to complete the training than GRID.

C. Simulation Evaluation

Settings. Our simulations are implemented on a physical server equipped with an Intel Core i9-10900 processor

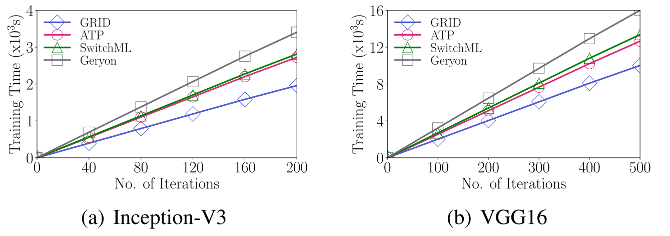


Fig. 9. Training time vs. no. of iterations.

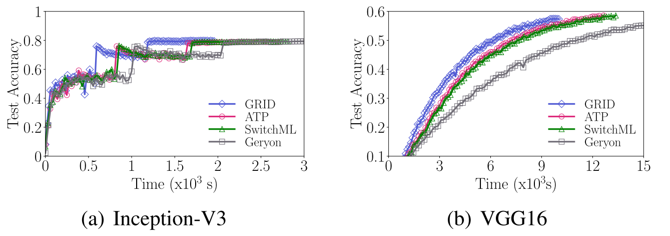


Fig. 10. Test accuracy vs. time.

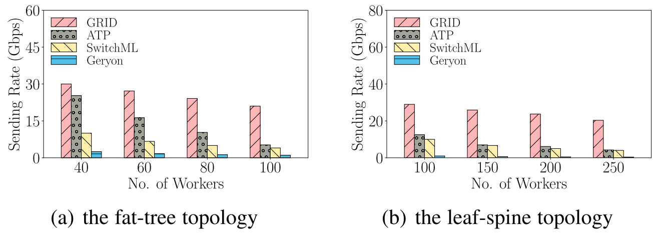


Fig. 11. Gradient sending rate vs. no. of workers in (a).

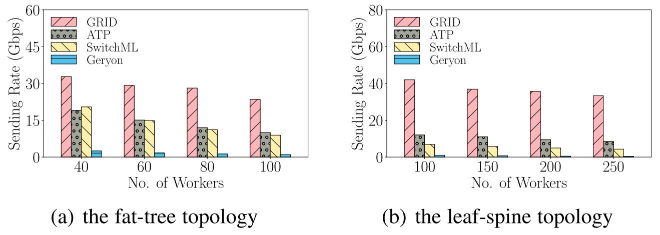


Fig. 12. Gradient sending rate vs. no. of workers in (b).

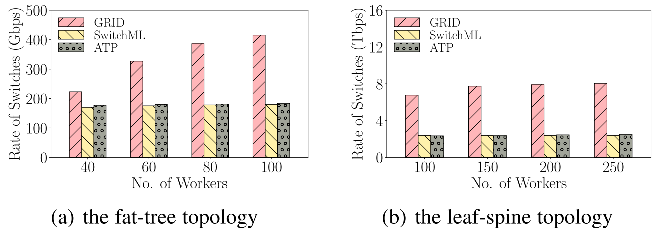


Fig. 13. Aggregation rate of switches vs. no. of workers in (a).

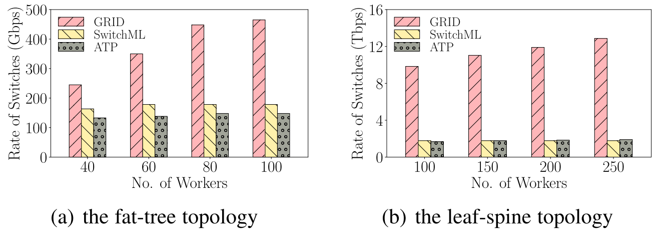


Fig. 14. Aggregation rate of switches vs. no. of workers in (b).

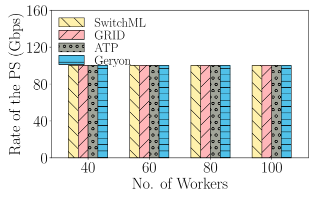
and 64GB RAM. We adopt the linear programming solver PuLP [58] to compute the routing policies. Note that, although mininet [70] supports replacing switches with bmv2 [71] software p4 switches, it faces a critical performance problem. After testing, we found that, when the scale of topologies increases 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 [72] also confirmed this conclusion. Therefore, we didn't choose to perform large-scale simulations through bmv2 and mininet, but through running the algorithm simulations.

We first obtain the gradient sending rates and aggregation policy by running the algorithm R-GRIA. Combining the gradient sending rate and aggregation policy, we can calculate the gradient aggregation rate of programmable switches and PS. Specifically, we accumulate the gradient sending rates of workers, whose aggregation nodes are programmable switches as the aggregation rate of programmable switches. Similarly, we accumulate the gradient sending rate of programmable switches and workers, which send gradients to the PS, as the aggregation rate of the PS.

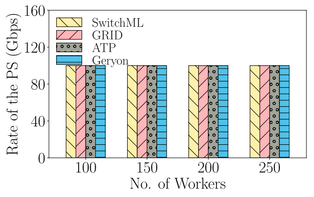
We select two practical topologies to verify the theoretical performance of GRID's routing algorithm. The first topology is the classical fat-tree topology [24], which contains 80 switches (32 edge switches, 32 aggregation switches and 16 core switches) and 192 servers. The second topology is a leaf-spine topology [25], which consists of 60 switches (30 spine switches and 30 leaf switches) and 500 servers. For both topologies, each element is connected with 100 Gbps links. The ingress bandwidth of the PS is set to 100 Gbps. Since the Wedge100BF-32x programmable switches contain 32 ports, each of which has a maximum bandwidth of 100 Gbps, we randomly set the processing capacity of programmable switches from 100 Gbps to 3.2 Tbps.

The simulations are performed under two scenarios. In the first scenario, we set the ToR switches in the fat-tree topology (leaf switches in the leaf-spine topology) as the programmable switches, similar to ATP [5]. Since the current datacenters deploy programmable switches as the ToR switches [5], [7], we evaluate the performance of GRID in large-scale datacenter programmable networks in this scenario, denoted by (a). In the second scenario, we randomly select 20% of the switches as programmable switches. Considering the popularity of programmable switches [73], [74], [75], we think that in the future programmable switches will be deployed throughout the datacenter network, not just as the ToR switches. Therefore, we evaluate the performance of GRID in general (programmable switches are not only deployed as ToR switches) large-scale programmable networks in this scenario, denoted by (b).

Comparison on gradient sending rate. This set of simulations gives the gradient sending rates comparison among these four algorithms and the results are shown in Figs. 11-12. We can see that, as the number of workers increases, the average gradient sending rate decreases, while GRID always

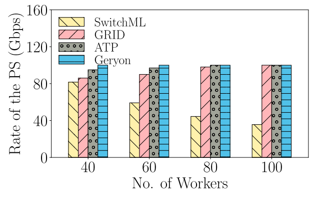


(a) the fat-tree topology

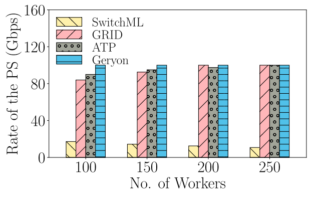


(b) the leaf-spine topology

Fig. 15. Aggregation rate of the ps vs. no. of workers in (a).



(a) the fat-tree topology



(b) the leaf-spine topology

Fig. 16. Aggregation rate of the ps vs. no. of workers in (b).

obtains the highest gradient sending rate compared with other alternatives. As shown in Fig. 11(a), given 40 workers, GRID outperforms the gradient sending rate by 0.19×, 2× and 11×, compared with ATP, SwitchML and Geryon. In scenario (a), the number of workers per rack (6 in the fat-tree topology and 16 in the leaf-spine topology) is determined. Due to the fixed routing paths of ATP and SwitchML, each programmable switch can only serve the workers in the same rack, resulting in underutilization. As the number of workers increases, the number of programmable switches increases so that the PS will receive more aggregated gradients from the switches. As a result, the ingress bandwidth of PS becomes the bottleneck of the scenario, limiting the gradient sending rates of workers. Since GRID decides the optimal aggregation nodes for workers, it can perform efficient in-network aggregation with fewer programmable switches than ATP and SwitchML, improving the gradient sending rates of workers. In the fat-tree topology in scenario (b), when the number of workers is 100, the average communication throughputs of GRID, ATP, SwitchML and Geryon are 23.5 Gbps, 10 Gbps, 8.9 Gbps and 1 Gbps, respectively. GRID outperforms the gradient sending rate by 1.35×, 1.64× and 22.5× compared with ATP, SwitchML and Geryon, respectively. The reason is that our algorithm selects proper gradient aggregation nodes for workers to achieve efficient in-network aggregation.

Comparison on gradient aggregation rate. This set of simulations is conducted to illustrate the aggregation rate performance of the programmable switches and the PS. The results are shown in Figs. 13-16. Since Geryon doesn't perform in-network aggregation, we omit it in Figs. 13-14. From Figs. 13-14 we can see that as the number of workers increases, GRID can consistently achieve the highest in-network aggregation throughput. Specifically, in Fig. 14(a), given 100 workers, the throughput of in-network aggregation of GRID, ATP and SwitchML are 465 Gbps, 148 Gbps and 178.45 Gbps, respectively. GRID improves the in-network aggregation throughput by 214% and 161.2% compared with ATP and SwitchML, respectively. Figs. 15-16 shows that GRID can utilize almost all ingress bandwidths of the PS for aggregation. For example, in Fig. 16(a), when the number of workers is 40, the aggregation throughput of the PS of SwitchML, GRID, ATP and Geryon are 81.6 Gbps, 86 Gbps, 95 Gbps and 100 Gbps, respectively. The reason is that, ATP and SwitchML ignore the importance of gradient routing, unable to utilize the aggregation capability of programmable switches efficiently. In contrast, GRID can select the optimal aggregation node for each worker to fully utilize the processing capacities of the programmable switches and the PS. Note that, in scenario (a), the ingress bandwidth of the PS is the bottleneck, so the aggregation rates of the PS are always 100Gbps.

Summary. From these simulation results, we can draw some conclusions. First, from Figs. 11-12 we can see that GRID achieves the highest gradient sending rate, which means GRID can speed up the distributed model training by performing efficient in-network aggregation. Second, Figs. 13-16 show that, by selecting the optimal aggregation nodes for workers, GRID can obtain a high aggregation rate of programmable switches and the PS.

VII. Conclusion

In this paper, we present a framework called **G**radient **R**outing with **I**n-network Aggregation for **D**istributed Training (GRID) to accelerate distributed model training in clusters. In the control plane, we formulate the GRIA problem, and propose an efficient randomized rounding based algorithm, named R-GRIA, to solve the problem. We further analyze the performance of R-GRIA. In the data plane, we design and implement in-network aggregation to ensure correct execution of routing policies in programmable switches. Extensive experimental and simulation results show that GRID can achieve high training throughput and speed up the distributed model training.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.
[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," 2018, *arXiv:1810.04805*.
[3] Z. Shuai, L. Yao, A. Sun, and T. Yi, "Deep learning based recommender system: A survey and new perspectives," *ACM Comput. Surv.*, vol. 52, no. 1, pp. 1–38, 2017.
[4] J. Liu et al., "Adaptive asynchronous federated learning in resource-constrained edge computing," *IEEE Trans. Mobile Comput.*, vol. 22, no. 2, pp. 674–690, Feb. 2023.
[5] C. Lao et al., "ATP: In-network aggregation for multi-tenant learning," in *Proc. NSDI*, 2021, pp. 741–761.
[6] 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.*, Aug. 2021, pp. 676–691.
[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] S. H. Hashemi, S. A. Jyothi, and R. Campbell, "TicTac: Accelerating distributed deep learning with communication scheduling," in *Proc. Mach. Learn. Syst.*, vol. 1, 2019, pp. 418–430.

[9] G. Zhao, H. Xu, Y. Zhao, C. Qiao, and L. Huang, "Offloading tasks with dependency and service caching in mobile edge computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 11, pp. 2777–2792, Nov. 2021.

[10] Y. Peng et al., "A generic communication scheduler for distributed DNN training acceleration," in *Proc. 27th ACM Symp. Operating Syst. Princ.*, Oct. 2019, pp. 16–29.

[11] M. Cho, U. Finkler, D. Kung, and H. Hunter, "BlueConnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy," in *Proc. Mach. Learn. Syst.*, vol. 1, Apr. 2019, pp. 241–251.

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

[13] A. F. Aji and K. Heafield, "Sparse communication for distributed gradient descent," 2017, *arXiv:1704.05021*.

[14] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "QSGD: Communication-efficient SGD via gradient quantization and encoding," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, pp. 1–12.

[15] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep gradient compression: Reducing the communication bandwidth for distributed training," 2017, *arXiv:1712.01887*.

[16] W. Wen et al., "TernGrad: Ternary gradients to reduce communication in distributed deep learning," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, pp. 1–11.

[17] M. Liu, S. Peter, A. Krishnamurthy, and P. M. Phothilimthana, "E3: Energy-efficient microservices on SmartNIC-accelerated servers," in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 363–378.

[18] J. Min et al., "Gimbal: Enabling multi-tenant storage disaggregation on SmartNIC JBOFs," in *Proc. ACM SIGCOMM Conf.*, Aug. 2021, pp. 106–122.

[19] Z. Yu et al., "Programmable packet scheduling with a single queue," in *Proc. ACM SIGCOMM Conf.*, Aug. 2021, pp. 179–193.

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

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

[22] B. Fortz and M. Thorup, "Internet traffic engineering by optimizing OSPF weights," in *Proc. 19th Annu. Joint Conf. IEEE Comput. Commun. Soc. (INFOCOM)*, vol. 2, Mar. 2000, pp. 519–528.

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

[24] 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.

[25] 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.

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

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

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

[29] 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)*, May 2020, pp. 8871–8875.

[30] 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.

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

[32] 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.

[33] 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.

[34] 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.

[35] R. McDonald, K. Hall, and G. Mann, "Distributed training strategies for the structured perceptron," in *Proc. Annu. Conf. North Amer. Chapter Assoc. Comput. Linguistics*, 2010, pp. 456–464.

[36] 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. AAAI Conf. Artif. Intell.*, vol. 33, 2019, pp. 1544–1551.

[37] 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.

[38] 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.

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

[40] A. Sergeev and M. Del Balso, "Horovod: Fast and easy distributed deep learning in TensorFlow," 2018, *arXiv:1802.05799*.

[41] P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations," *J. Parallel Distrib. Comput.*, vol. 69, pp. 117–124, Feb. 2009.

[42] Z. Cheng and Z. Xu, "Bandwidth reduction using importance weighted pruning on ring AllReduce," 2019, *arXiv:1901.01544*.

[43] 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.

[44] A. Jayarajan, J. Wei, G. Gibson, A. Fedorova, and G. Pekhimenko, "Priority-based parameter propagation for distributed DNN training," in *Proc. Mach. Learn. Syst.*, A. Talwalkar, V. Smith, and M. Zaharia, Eds. 2019, pp. 132–145.

[45] G. Wang, S. Venkataraman, A. Phanishayee, N. Devanur, J. Thelin, and I. Stoica, "Blink: Fast and generic collectives for distributed ML," in *Proc. Mach. Learn. Syst.*, vol. 2, 2020, pp. 172–186.

[46] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, "A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters," in *Proc. 14th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, 2020, pp. 463–479.

[47] R. Bhaskar, R. Jaiswal, and S. Telang, "Congestion lower bounds for secure in-network aggregation," in *Proc. 5th ACM Conf. Secur. Privacy Wireless Mobile Netw.*, Apr. 2012, pp. 197–204.

[48] M. A. Sharaf, J. Beaver, A. Labrinidis, and P. K. Chrysanthis, "TiNA: A scheme for temporal coherency-aware in-network aggregation," in *Proc. 3rd ACM Int. Workshop Data Eng. Wireless Mobile Access*, Sep. 2003, pp. 69–76.

[49] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *J. Mach. Learn. Res.*, vol. 18, no. 1, pp. 6869–6898, 2017.

[50] L. Mai et al., "Netagg: Using middleboxes for application-specific on-path aggregation in data centres," in *Proc. 10th ACM Int. Conf. Emerg. Netw. Exp. Technol.*, 2014, pp. 249–262.

[51] P. Costa, A. Donnelly, A. Rowstron, and G. O'Shea, "Camdoop: Exploiting in-network aggregation for big data applications," in *Proc. 9th USENIX Conf. Networked Syst. Design Implement.*, 2012, p. 3.

[52] *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

[53] *Nvidia Mellanox*. Accessed: Feb. 28, 2022. [Online]. Available: https://support.mellanox.com/s/productdetails/a2v1T000001JQCQQA4/sn4700

[54] *Netfpga*. Accessed: Feb. 28, 2022. [Online]. Available: https://netfpga.org/

[55] N. Gebara, M. Ghobadi, and P. Costa, "In-network aggregation for shared machine learning clusters," in *Proc. Mach. Learn. Syst.*, vol. 3, 2021, pp. 829–844.

[56] P. Bosshart et al., "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 87–95, Jul. 2014.

[57] R. Segal, C. Avin, and G. Scalosub, "SOAR: Minimizing network utilization with bounded in-network computing," in *Proc. 17th Int. Conf. Emerg. Netw. Exp. Technol.*, Dec. 2021, pp. 16–29.
[58] *Pulp*. Accessed: Jul. 20, 2021. [Online]. Available: https://pypi.org/project/PuLP/
[59] J. Dean et al., "Large scale distributed deep networks," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 25, 2012, pp. 1223–1231.
[60] M. Khani et al., "SiP-ML: High-bandwidth optical network interconnects for machine learning training," in *Proc. ACM SIGCOMM Conf.*, Aug. 2021, pp. 657–675.
[61] *Iftop*. Accessed: Apr. 14, 2022. [Online]. Available: http://www.ex-parrot.com/~pdw/iftop/
[62] *Intel P4 Studio SDE*. Accessed: Apr. 9, 2022. [Online]. Available: https://www.intel.com/content/www/us/en/products/docs/network-io/ethernet-programmable-switch/ica-1131-prospectus.html
[63] A. Paszke et al., "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 8024–8035.
[64] O. Russakovsky et al., "ImageNet large scale visual recognition challenge," *Int. J. Comput. Vis.*, vol. 115, no. 3, pp. 211–252, Dec. 2015.
[65] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2015, pp. 2818–2826.
[66] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2015, *arXiv:1409.1556*.
[67] A. Graves and J. Schmidhuber, "Framewise phoneme classification with bidirectional LSTM and other neural network architectures," *Neural Netw.*, vol. 18, no. 5, pp. 602–610, 2005.
[68] A. Krizhevsky, V. Nair, and G. Hinton. *Cifar-100 (Canadian Institute for Advanced Research)*. Accessed: Feb. 20, 2023. [Online]. Available: https://www.cs.toronto.edu/~kriz/cifar.html
[69] *The Wikitext Long Term Dependency Language Modeling Dataset*. Accessed: Apr. 9, 2022. [Online]. Available: https://www.salesforce.com/products/einstein/airesearch/the-wikitext-dependency-language-modeling-dataset/
[70] *An Instant Virtual Network on Your Laptop*. Accessed: Feb. 20, 2023. [Online]. Available: http://mininet.org
[71] *Behavioral Model Version 2 (BMV2)*. Accessed: Feb. 20, 2023. [Online]. Available: https://github.com/p4lang/behavioral-model
[72] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis, "In-network computation is a dumb idea whose time has come," in *Proc. 16th ACM Workshop Hot Topics Netw.*, Nov. 2017, pp. 150–156.
[73] D. Ding, M. Savi, G. Antichi, and D. Siracusa, "Incremental deployment of programmable switches for network-wide heavy-hitter detection," in *Proc. IEEE Conf. Netw. Softwarization (NetSoft)*, Jun. 2019, pp. 160–168.
[74] Y. Shi, M. Wen, and C. Zhang, "Incremental deployment of programmable switches for sketch-based network measurement," in *Proc. IEEE Symp. Comput. Commun. (ISCC)*, Jul. 2020, pp. 1–7.
[75] Z. Guo, W. Chen, Y.-F. Liu, Y. Xu, and Z.-L. Zhang, "Joint switch upgrade and controller deployment in hybrid software-defined networks," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 5, pp. 1012–1028, May 2019.

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.

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 articles 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 in several famous conferences.

Changbo Wu received the B.S. degree from the University of Electronic Science and Technology of China 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.

Jin Fang received the B.S. degree from the College of Computer Science and Electronic Engineering, Hunan University, in 2016. 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.

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.