ALEPH: Accelerating Distributed Training with eBPF-based Hierarchical Gradient Aggregation

Peng Yang, Hongli Xu, *Member, IEEE,* Gongming Zhao, *Member, IEEE,* Qianyu Zhang, Jiawei Liu
Chunming Qiao, *Fellow, IEEE*

Abstract—Distributed training includes two important operations: gradient transmission and gradient aggregation, which will consume massive bandwidth and computing resources. To achieve efficient distributed training, one must overcome two critical challenges: *heterogeneity of bandwidth resources* and *limitation of computing resources* among compute nodes. Existing architectures based on Parameter Server (PS) and All-Reduce (AR) fail to cope with these challenges because the PS will aggregate gradients from all workers and suffers from bandwidth bottlenecks, while AR intends to alleviate bandwidth bottlenecks at the PS, but the workers need to process many gradient packets thus can be overloaded. To address these shortcomings, we design a new distributed training system called ALEPH. In the control plane, ALEPH uses an efficient algorithm to group workers into clusters with different sizes so as to fully utilize heterogeneous bandwidth. We show that the proposed algorithm can achieve a good approximation performance. In the data plane, ALEPH leverages, for the first time, extended Berkeley Packet Filter (eBPF) programs to aggregate and forward gradient packets to reduce computation overhead. We show how to overcome several hurdles in using eBPF for distributed training. We implement ALEPH and evaluate its performance on a small-scale testbed and large-scale simulations. Experimental results show that ALEPH reduces training time by 20%-31% and increases bandwidth utilization by 88% compared with state-of-the-art frameworks.

Index Terms—*Distributed Training, Hierarchical Aggregation, eBPF, XDP, Datacenter.*

1 Introduction

In recent years, the renaissance of Deep Neural Networks (DNNs) has brought breakthroughs to several application domains, including computer vision [1], natural language processing [2] and recommendation systems [3]. To achieve satisfactory performance, deeper and more sophisticated DNNs are trained using larger datasets, which results in doubling training time every 3.4 months [4]. Paralleling and deploying DNN training tasks on multiple computing nodes, called *distributed DNN training* [5], [6], [7], is a practical method to accelerate training.

To perform distributed DNN training, there are two classic training architectures: Parameter Server (PS) [8] and All-Reduce (AR) [9]. In the PS architecture, computing nodes are classified into two types: parameter servers (PSs) and workers. During each training iteration, workers push their local gradients to PSs for aggregation and then pull the aggregated gradients from PSs. It is well known that the PS suffers from the bandwidth bottleneck [10], [11], as the bandwidth of PSs will be quickly exhausted with the increment of workers.

The All-Reduce (AR) architecture is proposed to avoid the bandwidth bottleneck in PSs, where the gradient aggregation is performed in a decentralized manner [12]. The most popular AR algorithm, ring AR [13], includes two phases: $scatter$ and $gather$. In the $scatter$ phase, each worker aggregates a part of gradients. Conversely, each aggregated part is transmitted to all workers in the $gather$ phase. Although the ring-based AR architecture conquers the bandwidth bottleneck in the PS, it requires gradients to go through more switches for gradient aggregation, resulting in a larger traffic volume [10], [14]. When a large number of gradient packets are processed by the network stack, the gradient aggregation and forwarding may overload the workers, and exhaust their computing resources [15]. This will, in turn, result in a high communication latency under the AR architecture.

To deal with the communication latency in AR architecture, the hierarchical gradient aggregation based on clustering was proposed [16], [17], [18], [19], [20]. Specifically, these studies first evenly partition all the workers into different clusters and then select a worker in each cluster as the cluster header to aggregate gradients. In each training iteration, all cluster headers will transmit the aggregated gradients to the PS for global aggregation. After that, the PS delivers aggregated gradients to workers through their cluster headers. As some gradients are aggregated in each cluster header, the congestion on the PS will be alleviated. When leveraging the AR architecture, each cluster header will perform gradient aggregation in parallel as well. In this way, the gradient aggregation based on clustering overcomes the disadvantages of PS and AR architectures and further reduces communication time.

However, it is not trivial to leverage gradient aggregation based on clustering to accelerate distributed training. In practice, distributed training will face two critical

- *P. Yang, H. Xu, G. Zhao, Q. Zhang, J. Liu are with the School of Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui, China, 230027, and also with Suzhou Institute for Advanced Research, University of Science and Technology of China, Suzhou, Jiangsu, China, 215123. E-mail: ypholic@mail.ustc.edu.cn, xuhongli@ustc.edu.cn, gmzhao@ustc.edu.cn, zqy2019@mail.ustc.edu.cn, liujiawei@mail.ustc.edu.cn.*
- *C. Qiao is with the Department of Computer Science and Engineering, State University of New York, Buffalo, NY, 14260, USA. E-mail: qiao@computer.org.*

challenges. The first challenge is *heterogeneity of bandwidth resources*. Existing hierarchical gradients aggregation often focuses on balanced clustering [16], [21], where workers are organized equally into clusters of the same size. However, the available bandwidth of these cluster headers may be heterogeneous [17], [22]. For example, Amazon Elastic Compute Cloud (EC2) offers over 500 instances [23], including different choices of CPU, GPU, bandwidth, *etc*. Affected by inventory availability and other factors, users may purchase instances with different specifications. Besides, the available bandwidth of workers may be heterogeneous due to contention with multiple jobs. Thus, the utilization of bandwidth resources is inadequate when the balanced clustering overlooks the heterogeneity of bandwidth resources.

The second challenge is *limitation of computing resources*. On the one hand, since distributed training with a large scale of workers, like 1088 workers training ResNet50 on the ImageNet dataset [19], is introduced to achieve better training performance, massive packets using TCP protocol need to be processed by the network stack. With limited computing resources, processing these packets by the network stack causes frequent interrupts and exhausts computing resources [24], [25], resulting in low aggregation throughput, which has been validated through experiments in Section 6.2. On the other hand, gradients will be aggregated multiple times in hierarchical aggregation, resulting in a higher demand for computing resources. Although Remote Direct Memory Access (RDMA) reduces the consumption of CPU resources, the lossless network (*e.g.*, Infiniband) required by RDMA is not common in commodity datacenters [26], [27]. Data Plane Development Kit (DPDK) is another way to reduce computation overhead, but this approach needs a redesign of the network stack and induces high CPU utilization even at low traffic load [28]. In conclusion, we should efficiently utilize the available bandwidth of workers and also take limited computing resources into consideration.

In order to overcome the above challenges, we propose a system called ALEPH, which achieves fast training speed by using both an efficient algorithm for clustering workers, and eBPF-based hierarchical gradient aggregation. More specifically, the key idea of ALEPH is two-fold: 1) To make the best utilization of heterogeneous bandwidth resources, we cluster workers based on the minimum dominating set (MDS) to form clusters of different sizes. 2) We implement the data plane leveraging the eBPF programs [29], which can bypass the kernel to significantly reduce the computing cost and process packets with low latency. To the best of our knowledge, this is the *first work* applying eBPF to gradient aggregation. Since eBPF is supported by the Linux Kernel, the proposed eBPF-based gradient packet processing method has wider applicability than RDMA and DPDK. In short, we make the following contributions in our paper:

- We first analyze two challenges for fast distributed training: *heterogeneity of bandwidth resources* and *limitation of computing resources*. To address the above challenges, we propose a system called ALEPH, which consists of a control plane and a data plane.
- In the control plane, we propose an MDS-based worker clustering algorithm to increase the gradient throughput, which overcomes the first challenge. The algorithm can achieve the approximation factor of $O(\log n)$ with high probability, where n is the number of workers.
- In the data plane, we design a pre-stack processing approach to aggregate and forward gradients, in order to be able to leverage the eBPF program to process packets with low latency and low computation overhead, which overcomes the second challenge.
- We conduct a small-scale testbed and a large-scale simulation using real-world models and datasets to show our mechanism can achieve fast distributed training compared with state-of-the-art solutions.

The rest of this paper is organized as follows. Section 2 introduces the related works. Section 3 presents our motivation and ALEPH overview. In Section 4, we describe the design of our control plane, which runs the worker clustering algorithm for hierarchical gradient aggregation. Section 5 gives the design of our data plane, which leverages eBPF programs to hierarchically forward and aggregate gradients. The experiment results are shown in Section 6, and we conclude this paper in Section 8.

2 Related Works

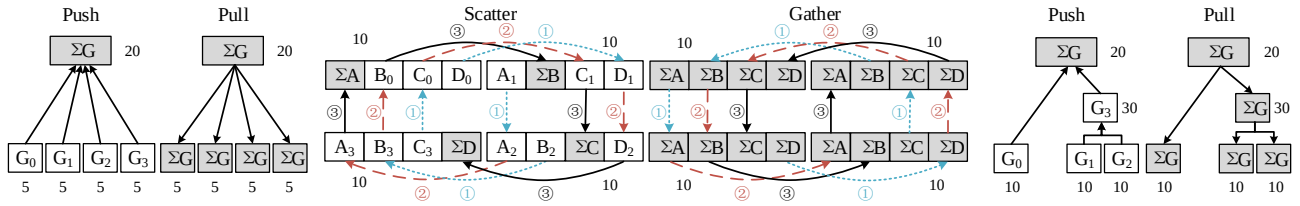
In datacenters, Parameter Server (PS) and All-Reduce (AR) are two classic training architectures. With the incremental number of compute nodes, recent studies show that the communication overhead has become the bottleneck of distributed training [30]. Many works are proposed to alleviate the bottleneck through hierarchical gradient aggregation or gradient aggregation acceleration.

2.1 Hierarchical Gradient Aggregation

One way of alleviating the communication bottleneck is the hierarchical gradient aggregation. For example, Geng *et al*. [16] propose HiPS, which combines the server-centric network topology with hierarchical gradient aggregation to achieve full use of network bandwidth. They also prove that HiPS better embraces server-centric network topologies, like BCube [26], DCell [31] and Torus [32]. Similarly, authors in [33] implement hierarchical gradient aggregation on 2D-Torus topology, which comprises of multiple rings in horizontal and vertical orientations. Jiang *et al*. [21] further improve the parallel efficiency of gradient aggregation by two-dimensional hierarchical ring-based AR architecture. Authors in [18] develop InHAD to perform an asynchronous distributed training based on hierarchical PS architecture, where an aggregator is allocated to aggregate gradients from training threads in the same worker. However, the above works often focus on the scenario of homogeneous bandwidth capacity, while the available bandwidth of workers is different because of hardware configurations and contention with multiple jobs. Thus, previous works may lead to long training time, especially when the available bandwidth of workers is heterogeneous.

2.2 Gradient Aggregation Acceleration

As complementary to the above works, gradient aggregation acceleration aims to reduce the cost and latency in processing gradient packets. With the increasing load of processing network packets in the network stack, the cost of computing resources on the workers grows as well. There



(a) Parameter Server (PS) (b) All-Reduce (AR) (c) ALEPH

Fig. 1: Illustration of gradient aggregation in PS, AR, and ALEPH. There are 4 workers with the available bandwidth of 10, 10, 10, and 30Gbps. (a): In the PS architecture, as the available bandwidth of the PS is 20Gbps, each worker transmits gradients with 5Gbps at most. (b): In the AR architecture, limited by the minimum bandwidth, each worker transmits gradients with 10Gbps. (c): In the ALEPH architecture, the worker with 30Gbps is selected as a cluster header and aggregates gradients from two workers. As a result, this worker occupies total 30Gbps bandwidth, where 20Gbps is used for the aggregation and 10Gbps is leveraged for global aggregation. Accordingly, other workers transmit gradients with 10Gbps, and the PS transmits gradients with 20Gbps.

Solution	Traffic Size	Transmission Rate	Time
PS	4.2Gb	5Gbps	0.84s
AR	6.3Gb	10Gbps	0.63s
ALEPH	4.2Gb	10Gbps	0.42s

TABLE 1: The comparison of traffic size each worker needs to send and receive, bidirectional transmission rate, and communication time of three solutions during one training iteration.

are three common technologies to bypass the kernel: RDMA [26], DPDK [28], and eBPF [29]. RDMA is able to offload the network stack to the hardware, and DPDK can implement the custom network stack. Consequently, many works accelerate gradient aggregation by RDMA and DPDK. First, some works use RDMA to accelerate training [34], [35], but RDMA requires network protocols like InfiniBand, iWarp, and RoCE [27]. Thus, professional NICs are required to support RDMA, which lacks universality in commodity datacenters. Second, authors in [36] use Intel DPDK to bypass the kernel and achieve fast gradient aggregation. Wu *et al*. [37] also leverage Intel DPDK technology to increase the bandwidth utilization within the GPU cluster at low computation overhead. However, DPDK processes packets via the polling mechanism and always consumes CPU resources even if there is no traffic. In addition, the DPDK passes layer-2 data directly to programs in user space, which is not conducive to the reuse of standard network stacks. As for eBPF, the recent work proposes BMC [38], which uses an eBPF-based program to accelerate key-value stores. Leveraging eXpress Data Path (XDP), eBPF programs can process layer-2 frames in a pre-stack way and reuse the standard network stack. To the best of our knowledge, existing works overlook the implementation of eBPF-based aggregation acceleration.

3 Motivation and System Overview

In this section, we give an example to illustrate the pros and cons of PS and AR architectures, which motivate our study. Then we present the overview and workflow of ALEPH.

3.1 A Motivating Example

We first present an example to illustrate the advantages and disadvantages of PS and AR architectures. According to different bandwidth sizes in Amazon EC2 instances, we analyze our example in different networks with 10Gbps, 20Gbps, and 30Gbps network bandwidths. Given 4 workers (*i.e.*, w_0, w_1, w_2, and w_3) running distributed DNN training tasks, the available network bandwidth of w_0, w_1, and w_2 is all 10Gbps, and w_3 has 30Gbps available bandwidth. Note that, gradient aggregation in synchronous communication sums gradients from all workers, which means the similar accuracy of PS, AR, and ALEPH after the same number of iterations. Thus, we just analyze the communication time for one training iteration under three solutions. Finally, the performance results are summarized in Table 1.

As shown in Fig. 1(a), workers under the PS architecture push their local gradients to the PS for aggregation and pull the aggregated gradients from the PS. G_i denotes local gradients from worker i. Assuming the gradient size is 4.2Gb (*e.g.*, VGG19 [39]), each worker needs to send 4.2Gb traffic for aggregation in each iteration. Meanwhile, the PS will receive gradients from four workers, *i.e.*, 16.8Gb traffic in total. Supposing that the available bandwidth of the PS is 20Gbps, it takes $\frac{16.8\mathrm{Gb}}{20\mathrm{Gbps}} = 0.84\mathrm{s}$ to receive gradients in the PS architecture. Thus, the transmission time is 0.84s during the $push$ phase. In fact, $push$ and $pull$ phases can be performed simultaneously, *i.e.*, the PS globally aggregates a segment of gradients and immediately sends this segment to all workers. As a result, the communication time is 0.84s during one training iteration in the PS architecture.

Fig. 1(b) shows the workflow of the AR architecture. According to the classical implementation of ring-based AR, it partitions gradients into 4 parts (*i.e.*, A, B, C, D), and each part is aggregated on an individual ring. Since each ring involves 3 steps and each step sends one part, shown as an arrow in Fig. 1(b), each worker will send and receive $\frac{3\times 4.2\mathrm{Gb}}{4} = 3.15\mathrm{Gb}$ traffic during the $scatter$ phase. During the $gather$ phase, each worker sends and receives gradients of the same size (*i.e.*, total 6.3Gb traffic). As a result, dominated by the slowest worker (with 10Gbps bandwidth), the communication time is $\frac{2\times 3.15\mathrm{Gb}}{10\mathrm{Gbps}} = 0.63\mathrm{s}$ during one training iteration in the AR architecture.

3.2 Our Intuition

The performance comparison of two classic architectures for distributed training is given in Table 1. In the PS architecture, each worker only processes 4.2Gb traffic during

Throughput	Consumption (eBPF)	Consumption (TCP)
5Gbps	0.3 CPU cores	3.5 CPU cores
10Gbps	0.6 CPU cores	7.5 CPU cores

TABLE 2: The comparison of computing resource consumption between eBPF programs and TCP programs under different aggregation throughput.

every iteration. However, due to the limited network bandwidth of the PS, each worker achieves a transmission rate of 5Gbps at most. Accordingly, the communication time of each iteration is 0.84s. In the AR architecture, workers achieve a transmission rate of 10Gbps via the decentralized mechanism. Since workers use bidirectional bandwidth to transfer gradients during both *scatter* and *gather* phases, each worker in the AR architecture transmits a relatively larger amount of traffic during one training iteration. Consequently, the communication time of each iteration is 0.63s. Since neither architecture achieves the ideal communication time, we need to propose a mechanism to deal with the bottlenecks in both solutions.

Following the above discussion, a question is *can we do better by combining the merits of PS and AR*. Clearly, we expect each worker to process less traffic volume like PS and better utilize available resources like AR. As shown in Fig. 1(c), considering that w_3 has sufficient bandwidth resources, we select w_3 to aggregate gradients from w_1 and w_2, and then w_3 transmits aggregated gradients to the PS. At the same time, w_0 delivers its gradients directly to the PS for global aggregation. In this way, the transmission rate of each worker achieves the same performance as AR (*i.e.*, 10Gbps), and the traffic size that workers deliver is as tiny as that in PS (*i.e.*, 4.2Gb). As shown in Table 1, it takes 0.42s for our solution to finish gradient aggregation during one training iteration. We leverage hierarchical aggregation to achieve theoretically optimal communication time.

Considering *limitation of computing resources*, it is challenging to implement the hierarchical gradient aggregation with low computation overhead. Traditionally, we use TCP to implement a simple gradient aggregation program, but we find that the program incurs significant resource overhead. Specifically, we deploy a TCP-based aggregation program on one worker with 8 CPU cores and 10Gbps bandwidth. We measure the aggregated traffic size per second (*i.e.*, aggregation throughput) using iftop [40] and the consumed computing resources using mpstat [41]. Besides, we deploy an aggregation program implemented by eBPF on the same worker and measure both the aggregation throughput and consumed computing resources. The results are shown in Table 2. For example, achieving the aggregation throughput of 10Gbps with a TCP program requires the utilization of 7.5 CPU cores. However, with the help of eBPF, the aggregation program is able to achieve the same throughput while reducing resource overhead by around 10 times. This means that we can leverage eBPF to accelerate gradient aggregation. Therefore, we design and implement the eBPF-based hierarchical gradient aggregation called ALEPH.

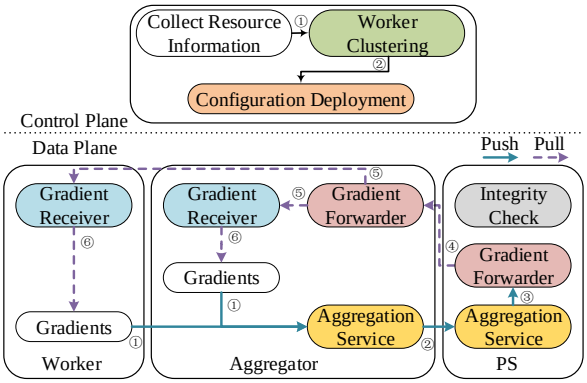


Fig. 2: Overview of ALEPH. In the control plane, *Worker Clustering* selects aggregators and workers into different clusters. *Configuration Deployment* informs each worker of the clustering result. In the data plane, *Aggregation Service* hierarchically aggregates gradients from workers, and *Gradient Forwarder* sends aggregated gradients back to workers. *Gradient Receiver* detects the receipt of complete gradients and notifies the training program. *Integrity Check* checks the lost packets.

3.3 System Overview and Workflow

ALEPH consists of a control plane and a data plane, as shown in Fig. 2. Note that in ALEPH, the cluster header is marked as an aggregator.

Control Plane is responsible for clustering workers according to resource constraints and informing all compute nodes about the IP address of the corresponding cluster header. To this end, the control plane is composed of two modules: *Worker Clustering* and *Configuration Deployment*. Depending on the available network bandwidth and computing resources of each worker, *Worker Clustering* selects the appropriate number of aggregators and notifies each worker to send its gradient to the specified aggregator. After that, *Configuration Deployment* informs each worker to initialize eBPF programs. We will describe the design of the control plane in Section 4.

Data Plane is responsible for aggregating and forwarding gradients. We leverage eBPF programs to process gradient packets with low computation overhead via four modules: *Aggregation Service*, *Gradient Forwarder*, *Gradient Receiver*, and *Integrity Check*. *Aggregation Service* on aggregators or the PS aims to aggregate and store gradients. *Gradient Forwarder* on aggregators or the PS forwards aggregated gradients back to workers. *Gradient Receiver* on workers provides notification for training programs when all aggregated gradients are received. *Integrity Check* on the PS is designed to ensure transmission reliability in case of congestion or error exceptions. Note that due to the limitations of the eBPF program, it is difficult to design the acceleration for sending gradients. We will elaborate on the design of the data plane for aggregating and forwarding gradients in Section 5.

Fig. 2 briefly describes the system workflow. In the beginning, the control plane clusters workers by *Worker Cluster* and informs all compute nodes to initialize eBPF programs via *Configuration Deployment*. When workers start training, gradients are sent to aggregators or the PS for hierarchical aggregation via *Aggregation Service* in steps ① and ②. Then, aggregated gradients are forwarded to workers by *Gradient Forwarder* in steps ③, ④ and ⑤. After that, each worker is

told to fetch aggregated gradients through *Gradient Receiver* in step ⑥. Besides, the data plane checks for packet loss by *Integrity Check* and retransmits gradients when detecting packet loss. Finally, the data plane loops the above steps during each iteration until training converges.

4 Control Plane Design

There are two modules in the control plane: *Worker Clustering* and *Configuration Deployment*. By formulating the problem of hierarchical aggregation, we propose an MDS-based algorithm for *Worker Clustering* (*M-WCHA*). Then, *Configuration Deployment* initializes the eBPF program.

4.1 Worker Clustering

System Model: In ALEPH, the PS architecture consists of a PS p and a worker set $W = \{w_1, w_2, ..., w_n\}$, where n is the number of workers in the system. With the evolution of Ethernet towards 400Gbps [42], [43], [44], the link bandwidth in datacenters usually will not become the bottleneck of distributed training, thus we ignore the link capability constraint as in [15]. We use $E = \{e_1, e_2, ..., e_{|E|}\}$ to denote the connection between two workers. For example, if e connects w_i and w_j, these two workers are able to communicate with each other. As a result, we obtain an undirect graph $G = (V, E)$, where $V = W \cup \{p\}$. $N(w)$ denotes the worker set (including w itself) that can directly communicate with w, that is $N(w) = \{w\} \cup \{w' | w' \text{ is adjacent to } w\}$.

For clustering workers, we mark the cluster header as an aggregator. In each cluster, the aggregator is responsible for aggregating gradients from all workers in the same cluster. After that, all aggregators send gradients to the PS for global aggregation. We use the variable D to denote the set of workers selected as aggregators. For each aggregator $w \in D$, we use V_w to represent the set of aggregator w and workers in the same cluster.

For each worker w, the available bandwidth and computing resource are denoted as B_w and C_w, respectively. We denote the bandwidth requirement of gradient aggregation as b (*e.g.*, 5Gbps) and computing requirement as c (*e.g.*, 10% CPU Usage) for workers. Considering the available bandwidth of workers, we initialize $b = \min\{B_{w_1}, ..., B_{w_n}\}$.

Problem Definition: To make better utilization of workers with limited bandwidth resources and computing resources, there are two constraints in the hierarchical aggregation (HA) problem.

1) *Bandwidth resource constraint*: If worker w is not an aggregator, it only sends local gradients for aggregation. With the sending rate $b = \min\{B_{w_1}, ..., B_{w_n}\} \leq B_w$, the bandwidth constraint is always satisfied. If w is an aggregator, it receives gradients from other workers in the same cluster and then transfers aggregated gradients to the PS. Since workers push and pull gradients simultaneously, w also receives gradients from the PS and forwards them to the corresponding workers. If w aggregates gradients from g_w workers, the required bandwidth resources are $(1+g_w)b$ for egress and ingress traffic. To satisfy the bandwidth resource constraint, it follows $(1 + g_w)b \leq B_w$, *i.e.*, $g_w \leq \frac{B_w}{b} - 1$.
2) *Computing resource constraint*: Since the aggregator w consumes extra computing resources for aggregation, it will aggregate gradients only from a limited number of workers. Assuming w aggregates gradients from g_w workers, the required computing resources are $g_w \cdot c$. To prevent exceeding computing capacity, we have $g_w \cdot c \leq C_w$, *i.e.*, $g_w \leq \frac{C_w}{c}$.

Algorithm 1 M-WCHA: MDS-based Worker Clustering for Hierarchical Aggregation

```
Initialize the variable, D = ∅
while W ≠ ⋃_{w∈D} V_w do
  Step 1: Updating Aggregation Capacity
  for worker w ∈ W do
    Update ḡ_w by Eq. (2) and ĝ_w by Eq. (3)
  Step 2: Selecting Candidate Aggregators (CAs)
  Initialize CA set D' = ∅ and worker set U by Eq. (4)
  for unclustered worker w ∈ U do
    for w' within 2 hops from w do
      if ĝ_w < ĝ_{w'} then
        goto Step 3
    Add w to D'
  Step 3: Clustering Workers for HA
  for worker w ∈ D' do
    Update α(w) by Eq. (5)
  for worker w' ∈ U − D' do
    Update β(w') by Eq. (6)
  for worker w ∈ D' do
    Find m and add w to D with probability 1/m
    if w is selected as an aggregator then
      Initialize V_w = {w} if V_w is empty
      for worker w' ∈ N(w) − ⋃_{w∈D} V_w do
        With probability α(w), add w' to V_w
```

In sum, we give the aggregation capacity $g_w = \min\{\frac{B_w}{b} - 1, \frac{C_w}{c}\}$ to denote the maximum number of workers that w can aggregate. Since the bottleneck of distributing training is caused by the congestion on the PS [10], [11], the objective of HA is to reduce the PS load, which means finding the minimum number of aggregators, namely $\min |D|$.

$$\min \quad |D|$$
$$S.t. \begin{cases} g_w = \min\{\frac{B_w}{b} - 1, \frac{C_w}{c}\}, & \forall w \in D \\ |V_w| - 1 \leq g_w, & \forall w \in D \\ V_{w_i} \cup V_{w_j} = \emptyset, & \forall w_i, w_j \in D \\ \bigcup_{w \in D} V_w = W \end{cases} \tag{1}$$

The first set of equations defines the aggregation capacity g_w of each worker as an aggregator. The second set of inequalities means that the number of workers in each cluster should not exceed the aggregation capacity of the aggregator. The third set of equations indicates that each worker is grouped into only one cluster. That is, the gradients of each worker are aggregated only by one aggregator. The final set of equations represents that all workers are grouped into clusters. Since the bandwidth of PS is consumed by aggregators and plenty of aggregators may cause the bandwidth bottleneck, our objective is to find the minimum number of aggregators, *i.e.*, $\min |D|$.

Algorithm Design: The HA problem becomes a typical minimum dominating set (MDS) problem [45] by relaxing two resource constraints. Thus, the MDS problem is a special

case of HA, which is NP-Hard. To solve the HA problem, we propose our MDS-based worker clustering algorithm including three steps.

In the first step, we determine the actual aggregation capacity $\bar{g}_w$ by Eq. (2) to denote the number of workers that can communicate with w, and compute the rounded aggregation capacity $\widehat{g}_w$ by Eq. (3) for fast comparison.

$$\bar{g}_w = \min\{g_w, |N(w) - \bigcup_{w \in D} V_w|\} \tag{2}$$

$$\widehat{g}_w = 2^x \geq \bar{g}_w, \text{smallest } x \in \mathbb{N} \tag{3}$$

The second step selects workers as candidate aggregators (CAs), where variable D' denotes the set of CAs. The unclustered worker set U is defined by Eq. (4).

$$U = W - \bigcup_{w \in D} V_w \tag{4}$$

When worker w has the largest rounded aggregation capacity among workers within 2 hops from w, we choose w as a CA and add w to D'.

In the third step, to compute the probability of a worker as an aggregator, we prepare two variables α and β. For each unclustered worker with one hop to w, $\alpha(w)$ is the probability that w aggregates its gradients. We use $N(w) - \bigcup_{w \in D} V_w$ to represent the set of unclustered workers adjacent to w. $\alpha(w)$ is the ratio of the aggregation capacity $\bar{g}_w$ to the number of unclustered workers adjacent to w and is expressed as

$$\alpha(w) = \begin{cases} 0, & \text{if } N(w) - \bigcup_{w \in D} V_w = \emptyset \\ \frac{\bar{g}_w}{|N(w) - \bigcup_{w \in D} V_w|}, & \text{otherwise} \end{cases} \tag{5}$$

$\beta(w')$ is the sum of the possibilities from all CAs adjacent to w'. Intuitively, we expect to select CAs whose adjacent workers all have small β as aggregators. We denote the set of CAs adjacent to w' as $A(w') = \{w | w \in N(w') \cap D'\}$. $\beta(w')$ is obtained as

$$\beta(w') = \sum_{w \in A(w')} \alpha(w) \tag{6}$$

For each CA w, we find the median value m of $\{\beta(w') | w' \in N(w) - \bigcup_{w \in D} V_w\}$ and select w as an aggregator with probability $1/m$. After that, we choose unclustered workers sending gradients to w with the probability of $\alpha(w)$. If no aggregator is selected in a round, all unclustered workers send gradients directly to the PS, and the algorithm stops. Otherwise, the algorithm will continue until all workers have been clustered.

Theorem 1. M-WCHA can achieve an approximation factor of $O(\log n)$ with high probability, where n is the number of workers.

Detailed proofs are shown in Appendix A.

4.2 Configuration Deployment

After aggregator selection, ALEPH will set up each computing node through three steps: notifying clustering results, creating gradient buffers, and initializing eBPF programs. In the first step, ALEPH uses a bitmap to identify workers. For each worker in one cluster, the aggregator marks each one with a local ID. The PS also marks each aggregator with a global ID. In the second step, ALEPH creates buffers (*i.e.*, eBPF maps) for caching gradients. ALEPH leverages each buffer to store 128 gradients, which are coincident with the gradient size of one packet. In the third step, each computing node compiles and loads eBPF programs at the XDP hook. Workers load the program for receiving gradients from the PS. Aggregators and the PS load programs for aggregating and forwarding gradients. Since eBPF programs bypass the kernel, ALEPH collects and maintains routing information like the IP address from all workers and the PS. When forwarding gradient packets, aggregators and the PS overwrite the IP address of packets according to the routing information. After the above steps, all workers in ALEPH have been set up and are ready to perform training.

5 Data Plane Design

This section first introduces the features of eBPF (Section 5.1) and then describes the challenges of implementing eBPF-based hierarchical gradient aggregation (Section 5.2). Finally, to overcome these challenges, we describe the detailed design of four modules in the data plane: *Aggregation Service*, *Gradient Forwarder*, *Gradient Receiver*, and *Integrity Check* (Sections 5.3-5.6).

5.1 Introduction of eBPF

eBPF [29], [46] is a revolutionary technology that allows sandboxed programs to run in the operating system kernel. Thus, eBPF can reprogram the behavior of the Linux kernel without modifying the kernel source. Some features of eBPF are as follows: (1) eBPF programs are event-driven. (2) eBPF introduces just-in-time compilation, and eBPF programs must be verified before being loaded into the Linux kernel. (3) eBPF programs leverage eBPF maps to share information in a wide set of data structures. These features have made eBPF to be applied in many applications, *e.g.*, BMC [38] and Cilium [47], and also motivate us to design eBPF-based gradient aggregation at the XDP [48] hook for low computation overhead.

5.2 Implementation Challenges

As eBPF programs at the XDP hook bypass the kernel and directly process layer-2 frames, ALEPH is expected to forward and process gradients with good performance. However, the features of eBPF bring new challenges when implementing gradient aggregation.

Interaction with eBPF programs: Training programs on workers obtain aggregated gradients from *Aggregation Service* (Section 5.3) on aggregators and the PS, so interaction with eBPF programs is inevitable. Since eBPF programs must run safely in the system kernel, eBPF only supports limited programmability. Accordingly, we encounter two challenges in achieving efficient interaction. 1) eBPF programs at the XDP hook do not support multicasting. The eBPF program is able to forward a gradient packet from one network interface to another, but is unable to copy the packet into multiple replicas and forward each one to a worker. To this end, we design the *Gradient Forwarder*

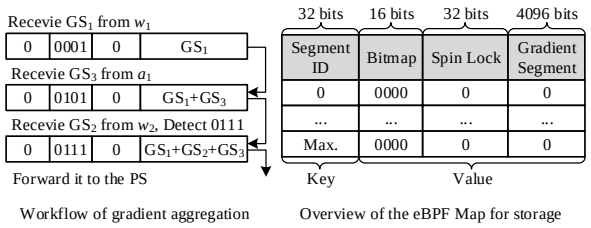


Fig. 3: Workflow of gradient aggregation and overview of gradient storage. Aggregator a_1 needs to aggregate gradients from workers in order of w_1, a_1, and w_2. When a_1 stores the gradient segment from w_2 and detects that the gradient aggregation is done, a_1 overwrites the packet from w_2 and forwards it to the PS. Otherwise, *Aggregation Service* will store aggregation results into the eBPF map.

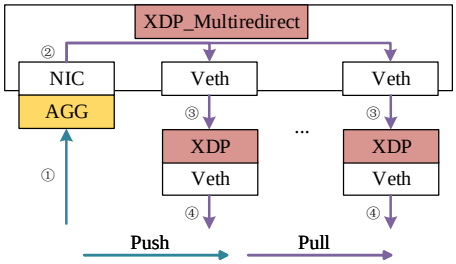


Fig. 4: Workflow of *Gradient Forwarder* for multicasting. When AGG (*Aggregation Service*) on the PS sums gradients from all aggregators, AGG sends aggregated gradient packets to XDP_Multiredirect. Then, packets are mechanically copied and sent to Veth pairs, where eBPF programs at XDP hook modify the IP address according to the corresponding aggregators.

module (Section 5.4) to implement the XDP-based gradient multicasting. 2) eBPF programs lack the function for process synchronization between the user and kernel space programs. For instance, when the training program waits for the reception of aggregated gradients, eBPF-based aggregation programs can not pass them directly to the training program. To overcome this, we design the *Gradient Receiver* module (Section 5.5) to notify training programs for fetching aggregated gradients. In conclusion, it is difficult to achieve a completely eBPF-based gradient aggregation due to the current lack of interaction with the eBPF program.

Reliable Transmission: Since our eBPF programs bypass the kernel, the stateful information about TCP is unavailable. Thus, it is difficult for eBPF programs to process TCP packets. To minimize the processing time, eBPF programs process packets using UDP protocol, which is also adopted in [38]. However, this mechanism may lead to unreliable transmission that affects the performance of distributed training. Consequently, we design *Integrity Check* to guarantee the reliable transmission of eBPF programs (Section 5.6) in ALEPH.

Below, we describe the four modules in the data plane to address the above challenges associated with using eBPF.

5.3 Aggregation Service

Aggregation Service is responsible for aggregating and storing gradients, and includes two components: Gradient Aggregation and Gradient Storage.

Gradient Aggregation filters packets corresponding to the ALEPH traffic and aggregates gradients of these packets. As shown in Fig. 3, we assume that an aggregator a_1 is responsible for aggregating gradients from workers in order of w_1, a_1, and w_2. When UDP packets are received by *Aggregation Service* of a_1, it checks whether this packet is sent for ALEPH or not. If not, this packet is processed by the network stack as usual. On receiving gradient packets from w_1, *Aggregation Service* stores these gradients into the eBPF map. As for packets from a_1, *Aggregation Service* aggregates gradients of these packets with stored gradients and stores the result of aggregation. When *Aggregation Service* receives packets from w_2 and detects that gradient segments from workers (*i.e.*, w_1, a_1, and w_2) have been aggregated, it stores the final aggregation result and passes the result to the PS.

Gradient Storage saves aggregated gradients with the eBPF maps. Specifically, eBPF maps are used to share collected information between eBPF programs and applications. As shown in Fig. 3, we split complete gradients into multiple segments and leverage the segment ID of split gradients as the key for searching the corresponding value in the eBPF map. The value is composed of a bitmap, a spinlock, and a segment of complete gradients. The bitmap records the source workers of aggregated gradients. The spinlock protects each eBPF map value from concurrent access because the eBPF map is shared by several cores. Note that XDP allows eBPF programs to process every received packet on each core, which can be realized by multi-queue support [49].

Note that, the impact of our eBPF program like *Aggregation Service* on other applications is negligible. Specifically, the eBPF program involves only a few lines of code that check each packet to determine its relevance to the ALEPH system. If the packet is irrelevant (*i.e.*., belongs to another application), it is forwarded to the protocol stack for normal processing. Therefore, the eBPF program introduces a negligible delay to other applications [50], [51].

5.4 Gradient Forwarder

Gradient Forwarder allows aggregators or the PS to forward packets to workers without the help of the network stack, which further lowers the consumption of CPU resources. Under a large-scale distributed training framework, delivering globally aggregated gradients from one PS to all workers will obviously increase the bandwidth pressure on the PS and lead to significant latency. Thus, ALEPH takes aggregators as relay nodes to forward aggregated gradients. As shown in Fig. 4, *Gradient Forwarder* on the PS receives aggregated gradients from *Aggregation Service* in step ① and forwards them to aggregators. Specifically, *Gradient Forwarder* copies one packet into multiple replicas via XDP_Multiredirect in step ②, modifies the IP address of each one via XDP programs in step ③, and then sends replicas to aggregators in step ④. Aggregators need to react differently when receiving packets from workers and the PS. Thus, we use a multicast signal to mark a packet that contains aggregated gradients from the PS. For simplicity, ALEPH reuses the bitmap in the eBPF map, where the PS sets each bit as 1 to represent the multicast signal. This signal

Algorithm 2 Integrity Check in ALEPH

```
1: Function Aggregation_Service_Integrity_Check()
2: Mask ← 1 << worker_id
3: for value in eBPF map do
4:    if value.bitmap & Mask ≠ Mask then
5:       Add value.segment_ID to lost_packets
6: Request for lost_packets
7:
8: Function Gradient_Receiver_Integrity_Check()
9: if no gradient fetch signal then
10:   for value in eBPF map do
11:      if value.bitmap ≠ 0xFFFF then
12:         Add value.segment_ID to lost_packets
13:   Request for lost_packets
```

also tells the aggregator to forward aggregated gradients to all workers in the same cluster.

5.5 Gradient Receiver

We design *Gradient Receiver* to notify the training program as soon as all aggregated gradients have been received. Since eBPF programs are unable to directly interact with blocked training programs which wait for aggregated gradients, ALEPH reuses the network stack to achieve the interaction with eBPF programs. Specifically, *Gradient Receiver* overwrites the destination port of the last gradient packet and inserts the gradient fetch signal. Then it passes the modified packet to a specific UDP listening port, which will trigger the training program to fetch gradients from eBPF maps. However, each packet that eBPF programs intercept is based on UDP protocol, leading to unreliable transmission. *Gradient Receiver* detects the last packet by counting the number of packets. If packets are lost, *Gradient Receiver* is unable to detect the last packet and fails to notify the training program. Thus, we design *Integrity Check* to ensure successful transmission of all gradients.

5.6 Integrity Check

Integrity Check provides a reliable transmission guarantee for *Aggregation Service* and *Gradient Receiver*, and Alg. 2 describes the pseudocode for *Integrity Check*. As shown in lines 1-6 of Alg. 2, the PS checks whether gradient packets from workers are lost or not, and asks for lost packets. Specifically, in each iteration of training, each worker sends a message to the PS when complete gradients have been transmitted. Leveraging the bitmap in the eBPF map, the PS records packet reception from each worker as one bit, where 1 stands for success and 0 for failure. After checking bitmaps, the PS sends requests to corresponding workers for lost packets. In a similar way, workers check whether aggregated gradients from the PS are lost or not, and request for the lost packets, which is shown in lines 8-13 of Alg. 2. Note that workers just verify the multicast signal to find which gradient packets are lost. Thus, ALEPH is able to resend all lost packets by *Integrity Check*.

6 Performance Evaluation

In this section, we first describe evaluation setups (Section 6.1). Then we use microbenchmark experiments to illustrate the advantages of eBPF programs in ALEPH (Section 6.2). We compare ALEPH using eBPF and M-WCHA with two popular distributed training frameworks on a small-scale testbed (Section 6.3). Finally, we show the performance of M-WCHA in a large-scale simulation scenario (Section 6.4).

6.1 Evaluation Setup

6.1.1 Benchmarks

ALEPH is compared with two popular distributed training frameworks: BytePS [7] and Horovod [5]. Specifically, BytePS and Horovod are based on PS and AR architectures, respectively, for inter-machine communication. Since eBPF programs in ALEPH leverage UDP protocol, we run two frameworks using TCP protocol for comparison. We choose two CNN models: ResNet50 with the size of 97MB [1] and VGG19 with the size of 548MB [52] as benchmark models to represent the compute-intensive and network-intensive workloads, respectively. Similar to [53], two models are trained with PyTorch, and the batch sizes of ResNet50 and VGG19 are both 64 per GPU. Each CNN model is trained with 150 epochs, and each epoch includes 98 iterations [54]. That is, the iteration is the number of batches needed to complete one epoch, and an epoch refers to one cycle through the full training dataset. We also choose BERT with the size of 438MB [55] as the Transformer model for the named entity recognition (NER) task [56], and the batch sizes are 16 per GPU [56]. We train the Transformer model with 10 epochs, and each epoch includes 84 iterations [54]. For PS architecture, workers and the PS are deployed on different machines, and each worker is equipped with one GPU. By default, we use the same number of workers with the same configuration to perform training in AR and PS architectures.

6.1.2 Performance Metrics

We adopt the following seven metrics to evaluate the performance of ALEPH: (1) the aggregation throughput; (2) the bandwidth usage of workers; (3) the average CPU usage; (4) the gradient communication time of one iteration; (5) the test accuracy; (6) the communication overhead; and (7) the gradient hops.

During the evaluation, we measure the traffic size per second received by aggregators or PSs for aggregation as *aggregation throughput*. The *bandwidth usage* of workers and PSs is monitored via iftop [40], a system monitor tool for network traffic. We use mpstat [41] to compute the *average CPU usage* for aggregating gradients. Like recent works [57], [58], the computation time and communication time are overlapped, where the communication starts as soon as a gradient is computed via backward propagation. Thus, we calculate the duration from the time a worker computes gradients till the time that the worker accepts aggregated gradients as the *gradient communication time of one iteration*. The percentage of correct predictions for the test data is called *the test accuracy*. The traffic size of gradients through links is accumulated as the *communication overhead*. Finally, the number of switches gradients pass through is recorded as *gradient hops*.

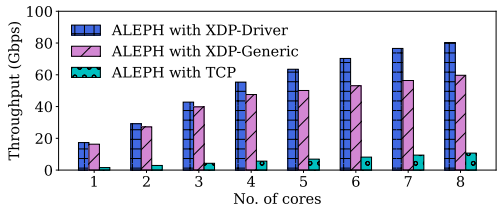


Fig. 5: Aggregation Throughput with Different No. of Cores

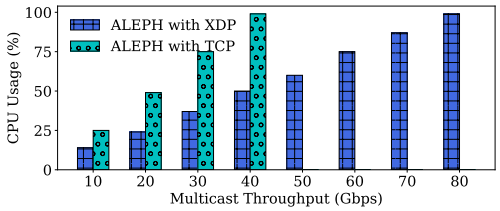


Fig. 8: Multicast Cost of XDP and TCP Programs

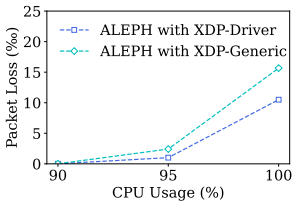


Fig. 6: Packet Loss vs. Avg. CPU Usage

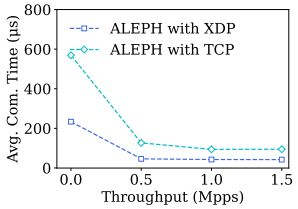


Fig. 7: Avg. Com. Time vs. Aggregation Throughput

6.2 Microbenchmark

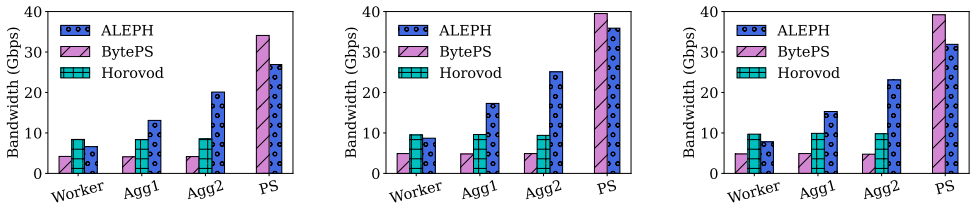
This section aims to measure the performance of the data plane in ALEPH, when ALEPH leverages XDP programs (eBPF programs at XDP hook) and TCP programs (the traditional way, denoted as ALEPH with TCP), respectively. To test the extreme performance, we equip two machines with one Mellanox 100GbE NIC respectively. We measure the CPU Usage, packet loss, and communication time for forwarding and aggregating gradients. Since the XDP program implemented via software realization includes two modes: network driver and system kernel, we test the performance of XDP-Driver programs (denoted as ALEPH with XDP-Driver) and XDP-Generic programs (denoted as ALEPH with XDP-Generic) separately.

1) Comparison on CPU Usage for Aggregation: In order to show the aggregation performance of ALEPH with XDP and TCP programs respectively, we measure the maximum aggregation throughput with the same number of CPU cores. We also evaluate how the performance of these programs scales with the number of CPU cores. The results are shown in Fig. 5. Specifically, the maximum aggregation throughput of ALEPH with XDP and TCP programs increases with the rising number of CPU cores, where the XDP-Driver program always achieves the highest throughput among the three programs. For example, on a single CPU core, the XDP-Driver program helps ALEPH achieve 17Gbps, but the TCP program only achieves 1.6Gbps. It means that the XDP-Driver program can improve $10\times$ aggregation throughput of ALEPH compared with the TCP program under the same number of CPU cores. With more than 3 CPU cores, ALEPH with XDP-Generic reaches the performance limit, and the aggregation throughput is capped at around 50Gbps. That is because processing packets by the network stack consumes more computing resources. Finally, ALEPH with TCP programs using 8 CPU cores can aggregate gradients at 10Gbps, while the XDP-Driver program and the XDP-Generic program in ALEPH achieve 80Gbps and 60Gbps, respectively. The reason is that XDP programs can process packets bypassing the kernel.

2) Comparison on Packet Loss: Although the design of *Integrity Check* is proposed to guarantee the successful transmission of gradients, we prefer to avoid retransmission caused by packet loss as much as possible. Since TCP protocol has congestion control capability, there is no serious packet loss. Thus, we mainly measure the packet loss of ALEPH with XDP programs under different CPU loads. Fig. 6 shows that the packet loss is low even when the CPU usage is over 90%. Without exhausting the network bandwidth, it may lead to a small packet loss rate of about 1% if the CPU usage is over 90%. When the CPU usage is about 95%, ALEPH with XDP-Driver loses 0.1% packets and ALEPH with XDP-Generic loses 0.2% packets. Besides, the XDP-Driver program in ALEPH loses packets without exceeding 1% at almost full CPU usage. It is reasonable that the XDP-Driver program performs better than the XDP-Generic program in ALEPH because programs running at the Linux kernel need extra operations like the skbuff allocation before processing network packets.

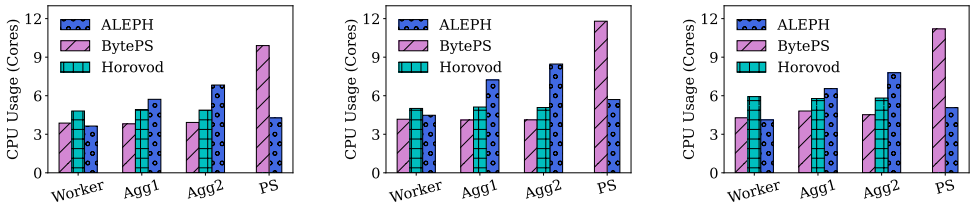
3) Comparison on Average Communication Time: Since ALEPH with XDP-Driver outperforms ALEPH with XDP-Generic in previous experiments, we implement ALEPH with XDP programs running in Driver mode by default (denoted as ALEPH with XDP). In this set of evaluations, we measure the average communication time for one gradient packet, where one packet contains 128 gradients. Specifically, the communication time is the elapsed time between sending a gradient and receiving it after aggregation. We use two workers to measure the average communication time, where one is responsible for sending and receiving gradients, and the other performs gradient aggregation. We leverage packets per second (pps) to show the aggregation throughput. Fig. 7 shows that as the throughput increases, the average communication time for one packet decreases. When aggregating gradients at 1pps, XDP programs in ALEPH achieve the time of $233\mu s$, and TCP programs in ALEPH obtain the time of $569\mu s$. As for 1.5Mpps, the time is $42\mu s$ for ALEPH with XDP and $94\mu s$ for ALEPH with TCP. The reason is that at low aggregation throughput, the average communication time for one packet is affected by the interrupt processing time.

4) Comparison on CPU Usage for Multicasting: We design the XDP-based multicast, which bypasses the kernel, to send aggregated gradients to all workers. To show the CPU resource consumption of gradient multicasting by XDP and TCP programs in ALEPH, we vary the throughput and measure the average CPU usage across 8 CPU cores. Fig. 8 shows that under the same throughput, ALEPH with XDP consumes fewer CPU resources compared to TCP-based multicast. For example, the maximum multicast



(a) ResNet50 (b) VGG19 (c) BERT

Fig. 9: Network Bandwidth Usage under Different Models



(a) ResNet50 (b) VGG19 (c) BERT

Fig. 10: CPU Usage under Different Models

throughput of the TCP program in ALEPH at nearly 100% CPU usage achieves 40Gbps. The XDP-based multicast can achieve 80Gbps at most with the full use of CPU resources. Compared with TCP programs in ALEPH, XDP programs obtain gradient multicasting with double throughput with the same CPU usage. The improvement of multicast performance is attributed to the design of bypassing the kernel. From these microbenchmark results, we can draw some conclusions. First, Fig. 5 and Fig. 8 show that, bypassing the Linux kernel, the CPU consumption of gradient aggregation and forwarding decreases significantly. Second, as shown in Fig. 6, XDP programs provide efficient communication with low packet loss. At last, from Fig. 7, we can see that the average communication time of ALEPH with XDP is lower than that of ALEPH with TCP. It means that ALEPH achieves lower CPU consumption and lower aggregation latency with the help of XDP programs.

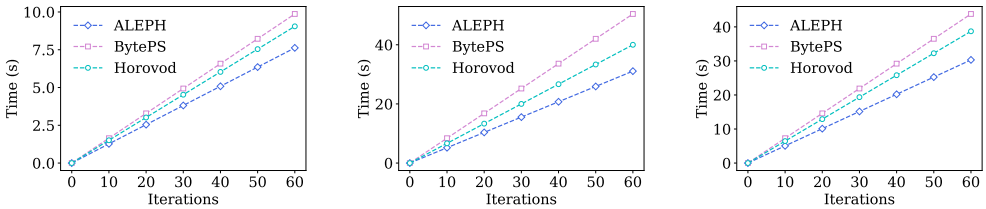
6.3 Testbed Evaluation

In this section, we first describe our testbed settings and then show different comparison results.

1) Testbed Settings: Similar to [34], we implement a testbed with the same size. Specifically, we use 9 machines to perform distributed training, including 8 workers and 1 PS. Each worker is equipped with one NVIDIA GeForce RTX 3080Ti GPU with NVIDIA driver version 515.48.07 and CUDA 11.7. Because the XDP program is designed to speed up the processing of network packets, it is unable to accelerate intra-machine communication among multiple GPUs. Therefore, we mainly consider each worker with one GPU. All machines are equipped with 44 cores of Intel(R) Xeon(R) Gold 6152 CPU@2.10GHz, 128GB RAM with Ubuntu 18.04, Linux kernel 5.14.21 and an Intel XL710 40GbE NIC. All machines are connected via a Mellanox 16x40Gbps Ethernet switch. To simulate our motivating example, we limit the network bandwidth of nine machines: one worker with 30Gbps bandwidth, two workers with 20Gbps bandwidth, five workers with 10Gbps bandwidth, and the PS with 40Gbps bandwidth.

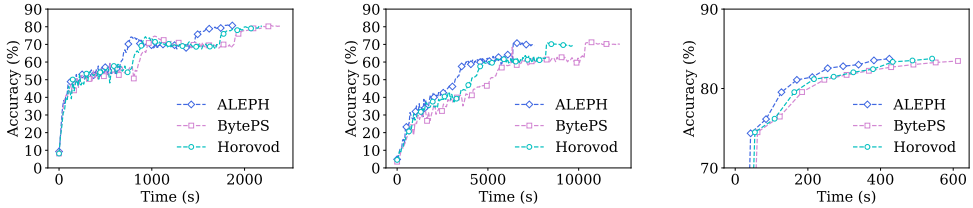
2) Comparison on Bandwidth Usage: In this set of experiments, we run tests to measure the network bandwidth usage of machines in different solutions. After running the M-WCHA algorithm, workers with 20Gbps bandwidth are selected as aggregators called Agg1, and each Agg1 aggregates gradients from one worker with 10Gbps bandwidth. The worker with network bandwidth of 30Gbps is also selected as the aggregator called Agg2, which is responsible for aggregating gradients from two workers with 10Gbps bandwidth. The last worker with 10Gbps bandwidth directly sends gradients to the PS. As shown in Fig. 9, workers and the PS in ALEPH achieve efficient utilization of network bandwidth. When training ResNet50, workers in BytePS only send gradients at 4.2Gbps, while workers in Horovod send gradients at 8.4Gbps. In ALEPH, workers transfer gradients at 6.6Gbps, and the throughput of the Agg1, Agg2, and PS achieves 13.1Gbps, 20.1Gbps, and 26.9Gbps, respectively. Finally, the bandwidth utilization of all workers in ALEPH increases by 136% and 18% compared with BytePS and Horovod in training ResNet50. The bandwidth utilization in ALEPH increases by 163% and 36% in training VGG19, respectively. When training BERT, ALEPH increases the bandwidth utilization by 141% and 19%, respectively. That is because ALEPH makes better use of workers with sufficient bandwidth to alleviate the congestion in the PS.

3) Comparison on CPU Usage: In this set of experiments, we run tests to measure the average CPU usage of each machine in different solutions. As shown in Fig. 10, machines in ALEPH achieve relatively low CPU consumption. For example, the Agg2 in ALEPH consumes 6.8 CPU cores for training ResNet50, 8.5 CPU cores for training VGG19, and 7.8 CPU cores for training BERT. The same machine in BytePS consumes 3.9 CPU cores while training ResNet50, 4.1 CPU cores while training VGG19, and 4.5 CPU cores while training BERT. Although the CPU usage of the Agg2 in ALEPH increases by nearly 50% compared with that in



(a) ResNet50 (b) VGG19 (c) BERT

Fig. 11: Communication Time under Different Models



(a) ResNet50 (b) VGG19 (c) BERT

Fig. 12: Accuracy under Different Models

BytePS, it achieves triple throughput. Besides, other workers in ALEPH consume lower CPU resources by 6% and 11% compared with BytePS and Horovod when the throughput is almost the same. The reason is that XDP programs consume low CPU resources compared with TCP programs.

4) Comparison on Training Performance: To estimate the performance of distributed model training, we compare the communication time of two models in different solutions. First, as shown in Fig. 11, the communication time per iteration by ALEPH is less than the time by BytePS and Horovod. Specifically, the communication time per iteration by BytePS, Horovod, and ALEPH is 0.17s, 0.15s, and 0.12s, respectively, for training ResNet50. The communication time per iteration by BytePS, Horovod, and ALEPH is 0.84s, 0.78s, and 0.58s, respectively, while training VGG19. When training BERT, the communication time per iteration by BytePS, Horovod, and ALEPH is 0.73s, 0.65s, and 0.51s. It means that compared with BytePS and Horovod, the communication time in ALEPH decreases by 29% and 20% in training ResNet50, decreases by 31% and 26% in training VGG19, and decreases by 30% and 22% in training BERT. Second, as shown in Fig. 12, ALEPH achieves a higher training accuracy with the same communication time compared with BytePS and Horovod. For example, ALEPH finishes training and obtains an accuracy of 80.4% when training ResNet50. Meanwhile, BytePS obtains an accuracy of 69.8% and Horovod obtains an accuracy of 78.1%. That is because ALEPH reduces communication time via efficient utilization of resources and low latency of aggregation.

From the above testbed results, we draw some conclusions. First, Fig. 9 shows that when selecting workers with sufficient bandwidth as aggregators, the bandwidth usage of workers in ALEPH can increase by at least 18% compared with other solutions. Second, as shown in Fig. 10, workers achieve lower CPU usage compared with BytePS and Horovod. At last, from Figs. 11-12, we will see that the communication time in ALEPH has been obviously reduced. It means that, by leveraging the MDS-HA algorithm and XDP programs, ALEPH achieves fast distributed training.

6.4 Simulation Evaluation

In this section, we first describe our simulation settings and show the advantages of our algorithm through extensive comparison results.

1) Simulation Settings: We simulate the communication through mininet tool [59] on a physical machine with an Intel Core i9-10900 processor and 64GB RAM. To estimate the performance of ALEPH, we select two practical topologies that are commonly adopted in the datacenter. The first topology is a three-layer fat-tree topology [60], which contains 16 core switches, 32 aggregation switches, 32 edge switches, and 320 servers. The second topology is a two-layer leaf-spine topology [61], which consists of 30 spine switches, 30 leaf switches, and 300 servers. Each worker is equipped with a 10Gbps NIC and transfers the traffic of 97MB (same as the gradient size of ResNet50) during each training iteration. We set the capacity of uplinks and downlinks in the proportion of 1:1 to provide a non-blocking network. To simulate network dynamics in the datacenter, we configure the available network bandwidth of workers with a normal distribution probability. Specifically, the average available bandwidth of workers is 5Gbps, and the default standard deviation is 0.1 [62]. The available bandwidth of each worker is set as $ratio \times$10Gbps, where $ratio \in [0.1, 0.9]$ is achieved according to the probability of normal distribution. We select 100 servers as workers and 10 servers as PSs to perform one training task, where each PS can provide 10Gbps for gradient aggregation. Note that, our algorithm aims to find the minimum number of cluster headers by considering the bandwidth and computing resources of workers. Therefore, the results of the clustering algorithm are not influenced by the number of PSs.

2) Comparison on Bandwidth Usage: We vary the standard deviation of the normal distribution to estimate the influence on bandwidth usage of workers and PSs. The results are shown in Fig. 13. When the available network bandwidth is heterogeneous, ALEPH achieves the best usage of network bandwidth among the three solutions. While the standard deviation is 0 (*i.e.*, homogeneous bandwidth),

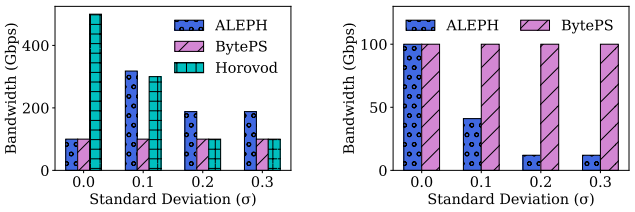


Fig. 13: Bandwidth Usage vs. Standard Deviation

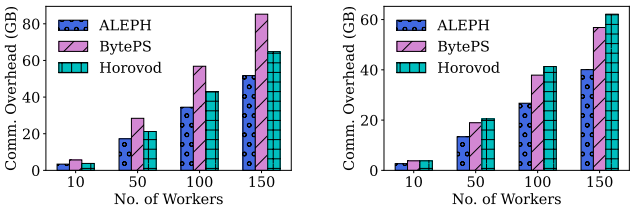


Fig. 14: Communication Overhead vs. No. of workers

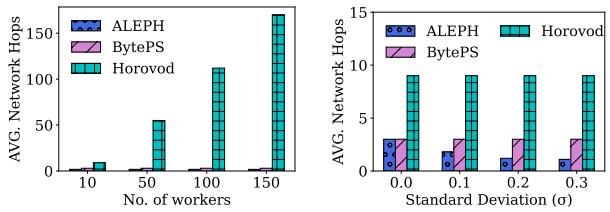


Fig. 15: AVG. Gradient Hops vs. No. of Workers

Fig. 16: AVG. Gradient Hops vs. Standard Deviation

Horovod obtains the highest utilization of network bandwidth, because AR is proved to be optimal for scenarios of homogeneous bandwidth [9]. With the standard deviation increasing, ALEPH outperforms other solutions. When the standard deviation is 0.3, the bandwidth usage of all workers in ALEPH, BytePS, and Horovod is 188Gbps, 100Gbps, and 100Gbps, respectively. Besides, the bandwidth usage of all PSs in ALEPH and BytePS is 12Gbps and 100Gbps. On the one hand, ALEPH increases the utilization of network bandwidth by 88% compared with other solutions. On the other hand, ALEPH decreases the pressure on the bandwidth of PSs by 88% compared with BytePS. The reason is that ALEPH makes full use of the workers with sufficient bandwidth to alleviate the congestion in PSs.

3) Comparison on Communication Overhead: In this set of simulations, we show the communication overhead of three solutions in fat-tree and leaf-spine topologies. Fig. 14 shows that ALEPH achieves less communication overhead compared with other solutions. Given 150 workers in the fat-tree topology, the communication overhead of ALEPH, BytePS, and Horovod is 51.7GB, 85.3GB, and 64.8GB, respectively. ALEPH decreases the communication overhead by 39% and 20% compared with BytePS and Horovod, respectively. Since ALEPH selects workers with one gradient hop for gradient aggregation, the communication overhead can be greatly reduced.

4) Comparison on Gradient Hops: We measure the average gradient hops of three solutions with different numbers of workers and varying standard deviations. In Fig. 15, the average number of gradient hops in centralized architectures (*i.e.*, ALEPH and BytePS) are not affected by the number of workers, because workers send gradients directly to PSs. In Fig. 16, given 10 workers, ALEPH achieves fewer hops compared with other solutions. When the standard deviation is 0.3, the average number of gradient hops in ALEPH, BytePS, and Horovod are 1.1, 3, and 9. ALEPH reduces the hop number by 63% and 88% compared with BytePS and Horovod. Since gradients are allowed to be aggregated at most twice, the latency caused by multiple gradient hops can be significantly reduced.

From these simulation results, we can draw some conclusions. First, Fig. 13 shows that compared with other solutions, ALEPH achieves better utilization of available network bandwidth and reduces communication congestion in PSs. Second, in Figs. 14-16, via hierarchical aggregation based on worker clustering, ALEPH performs the distributed training with low communication overhead and a small number of gradient hops.

7 Discussion

ALEPH also can support the multi-PS scenario. For example, the workflow of ALEPH with q PSs is as follows: In the control plane, ALEPH first clusters the workers using an algorithm and initializes the corresponding eBPF programs. In the data plane, each worker within a cluster sends its gradients to the respective aggregator. Once the aggregator within each cluster completes gradient aggregation, each aggregator will split aggregated gradients into q parts and send each part to one PS. The splitting ratio for each gradient part depends on the available bandwidth of each PS relative to the total available bandwidth of all PSs. This enables ALEPH to effectively tackle the challenge of heterogeneous bandwidth resources of PSs.

8 Conclusion

In this paper, we have proposed a new architecture that aims to accelerate distributed training with eBPF-based hierarchical gradient aggregation called ALEPH, and provided a complete design that includes a control plane and a data plane. In the control plane, we have proposed an MDS-based worker clustering algorithm for hierarchical aggregation, called M-WCHA, to fully utilize the heterogeneous bandwidth resources, and showed that the algorithm achieves a good approximation performance. In the data plane, we have designed four modules to enable us to overcome the technical hurdles so as to be able to apply the eBPF programs for the first time to gradient aggregation, in order to reduce the consumption of the limited computing resources at the workers. Extensive testbed-based and simulation-based experimental results have shown that ALEPH can significantly accelerate distributed training. In our future work, we will improve the performance of ALEPH in scenarios with heterogeneous computing resources and large-scale distributed training.

ACKNOWLEDGMENT

The corresponding author of this paper is Hongli Xu. This research received partial support from several funding sources, including the National Science Foundation of China (NSFC) under Grants 62372426, 62132019, and 62102392; the National Science Foundation of Jiangsu Province under Grant BK20210121; the Fundamental Research Funds for the Central Universities; and the Youth Innovation Promotion Association of the Chinese Academy of Science (2023481).

APPENDIX A
PERFORMANCE ANALYSIS

We analyze the approximation performance of the M-WCHA algorithm.

Theorem 2. M-WCHA can guarantee that each aggregator aggregates gradients without exceeding its bandwidth and computing capacities.

Proof: According to each worker's bandwidth and computing capacities, we obtain that the worker w can aggregate gradients from g_w workers at most. In line 23 of Alg. 1, we select unclustered workers sending gradient to the aggregator with the probability of $\alpha(w)$, which is the ratio of aggregation capacity to the number of unclustered workers with one hop. Thus, M-WCHA guarantees the resource capacity constraint. □

Theorem 3. M-WCHA guarantees that each worker sends its local gradients to only one aggregator, *i.e.*, the third set of equations in Eq. (1).

Proof: In lines 22-23 of Alg. 1, M-WCHA only selects a worker from the unclustered worker set. Namely, when a worker is selected to send gradients to one aggregator in a round, this worker will be removed from the unclustered worker set and be never selected as an aggregator in the following rounds. As a result, gradients from each worker are only aggregated by one aggregator. □

Before analyzing the expected number of aggregators, we introduce some notations and three lemmas. In the graph $G=(V,E)$, we use $G'=(W',E')$ as the subgraph of G. In one round, W' is a set including all CAs and all unclustered workers with one hop to one of these CAs, and E' is a set of edges whose endpoints consist of a CA and an unclustered worker. To estimate the number of aggregators, we calculate the clustering cost when selecting a worker w' to send its gradients to an aggregator w. We define $\phi(w')=1/\widehat{g}_w$ as the clustering cost.

Lemma 4. If two CAs are connected by a path in G', they have the same rounded aggregation capacity.

Proof: By the definition of E', each edge only connects one CA and one unclustered worker. Since each CA has the maximum rounded aggregation capacity within two hops, any two CAs lying on a path must have the same rounded aggregation capacity. Otherwise, it violates the definition of E'. It is obvious that all CAs in G' have the same rounded aggregation capacity. □

Lemma 5. If an unclustered worker w' is selected to send gradients to an aggregator with the clustering cost $\phi(w')$, $\sum_{w'\in V}\phi(w')\leq H_\Delta|OPT|$, where OPT is the minimum aggregator set and H_Δ is a constant.

Proof: According to Lemma 4, all CAs that can aggregate gradients from w' have the same rounded aggregation capacity. Thus, we choose any CA as an aggregator and do not change the value of $\phi(w')$. Following the analysis of greedy algorithms [63], [64], we estimate an upper bound on our algorithm.

Assuming the minimum aggregator set is OPT, one aggregator $w\in OPT$ aggregates gradients from workers in V_w with the size of k. To get an ordered sequence $w'_1,w'_2,\cdots,w'_k$, we sort all workers in V_w such that $1\leq i<j\leq k$, where w'_i is selected before w'_j. We compute $\phi(w'_i)=\frac{1}{\widehat{g}_w}\leq\frac{1}{\bar{g}_w}\leq\frac{1}{k-i+1}$. It follows

$$\begin{aligned}\sum_{w'\in V_w}\phi(w')&=\sum_{i=1}^{k}\phi(w'_i)=\frac{1}{k}+\frac{1}{k-1}+...+\frac{1}{1}\\&=H_k\leq H_\Delta\end{aligned}\tag{7}$$

where Δ is $\max\{g_w,d_w|w\in V\}+1$, and d_w is the number of workers with one hop to w. When all workers have been clustered, we can obtain

$$\sum_{w'\in W}\phi(w')\leq\sum_{w\in OPT}\sum_{w'\in V_w}\phi(w')\leq H_\Delta|OPT|\tag{8}$$

□

Lemma 6. If S is the set of CAs added to D and Z denotes the total cost for clustering workers in a round, we have the expectation $\mathbb{E}[|S|]\leq 8\mathbb{E}[Z]$.

Proof: For aggregator $w\in S$, we sort the element $w'\in V_w$ according to the $\beta(w')s$ in non-increasing order. Let $F(w)$ and $L(w)$ respectively denote the set of the first $\lceil V_w/2\rceil$ workers and the last $\lceil V_w/2\rceil$ workers in this sorted order. Since $\widehat{g}_w=2^x\geq\bar{g}_w\geq 2^{x-1}$ and $|V_w|\geq\bar{g}_w$, we have $|V_w|\geq\widehat{g}_w/2$ for $\forall w\in S$. As $|V_w|\leq 2|L(w)|$, we obtain

$$\begin{aligned}|S|&=\sum_{w\in S}1\leq\sum_{w\in S}|V_w|\frac{2}{\widehat{g}_w}\\&\leq 2\sum_{w\in S}\sum_{w'\in V_w}\phi(w')\\&\leq 4\sum_{w\in S}\sum_{w'\in L(w)}\phi(w')\end{aligned}\tag{9}$$

For $w'\in U$, we use $t(w')$ to denote the number of CAs added to D at the end of this round when $w'\in L(w)$ and $w\in S$. Otherwise, we set $t(w')=0$. We now have

$$|S|\leq 4\sum_{w\in S}\sum_{w'\in L(w)}\phi(w')=4\sum_{w'\in U}\phi(w')t(w')\tag{10}$$

In a given round, $\phi(w')$ is fixed and we have the expectation

$$\mathbb{E}[t(w')|t(w')>0]=\frac{\mathbb{E}[t(w')]}{\mathbf{Pr}[t(w')>0]}\tag{11}$$

By substituting $t(w')$ in Eq. (10), we can obtain

$$\mathbb{E}[|S|]\leq 4\sum_{w'\in U}\phi(w')\mathbf{Pr}[t(w')>0]\mathbb{E}[t(w')|t(w')>0]\tag{12}$$

We define $H=\{w|w'\in L(w)\}$. For $w\in H$, $h(w)$ represents the probability that w is added to D. As $w'\in L(w)$ and $h(w)\leq 1/m\leq 1/\alpha(w')\leq 1/|H|$ for $w\in H$, we get

$$\mathbb{E}[t(w')|t(w')>0]$$

$$
\begin{aligned}
&= \sum_{w\in H} \mathbf{Pr}[w \text{ is added to } D | t(w') \geq 1] \\
&= \sum_{w\in H} \frac{\mathbf{Pr}[w \text{ is added to } D]}{\mathbf{Pr}[t(w') \geq 1]} \\
&= \frac{\sum_{w\in H} h(w)}{1-\prod_{w\in H}(1-h(w))} \\
&\leq \frac{\sum_{w\in H} h(w)}{\sum_{w\in H} h(w) - \sum_{w_1,w_2\in H, w_1\neq w_2} h(w_1)h(w_2)} \\
&\leq \frac{\sum_{w\in H} h(w)}{\sum_{w\in H} h(w) - \sum_{w\in H} h(w)/2} = 2
\end{aligned} \tag{13}
$$

Thus, we replace the bound on $\mathbb{E}[t(w')|t(w')>0]$ in Eq. (12).

$$
\mathbb{E}[|S|] \leq 8 \sum_{w'\in V_w} \phi(w') \mathbf{Pr}[t(w')>0] \leq 8\mathbb{E}[Z] \tag{14}
$$

Finally, we prove the theorem by Eq. (14). □

Theorem 7. M-WCHA obtains the expected size of the aggregator set by $8H_\Delta|OPT|$, where OPT is the minimum aggregator set and H_Δ is a constant obtained in Eq. (7). M-WCHA achieves an approximation factor of $O(\log n)$ with high probability, where n is the number of workers.

Proof: The set of aggregators selected in round i are denoted as S_i, and the total cost for clustering workers in this round is Z_i. Hence, we achieve the expected number of aggregators as

$$
\sum_i \mathbb{E}[|S_i|] \leq 8 \sum_i \mathbb{E}[Z_i] = 8 \sum_{w'\in V} \mathbb{E}[\phi(w')] \leq 8H_\Delta|OPT| \tag{15}
$$

To explore the bound with high probability, we view aggregator selections in each round as a set of independent selections. We set the probability for one worker w as an aggregator in round i as $h_i(w)$. $h_i(w)$ is a random variable, but will be affected by the consequence of previous selections. If a worker w is selected as an aggregator in round i, $h_j(w)$ is zero for any following round j, with $j > i$. Hence, aggregator selections are not equivalent to a set of independent Bernoulli trials. However, if we focus on any specific execution of random selections, the sum of the probabilities is upper bounded by $H_\Delta|OPT|$, where Δ is $\max\{g_w, d_w | w \in V\} + 1$, where d_w is the number of workers with one hop to w and g_w is the aggregation capacity of w. We leverage a Chernoff-type argument to achieve the number of aggregators by an approximation factor of $O(\log n)$ with high probability [45], [65], [66]. □

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, pp. 770–778.

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[3] P. Covington, J. Adams, and E. Sargin, "Deep neural networks for youtube recommendations," in *Proceedings of the 10th ACM conference on recommender systems*, 2016, pp. 191–198.

[4] D. H. Dario Amodei, "Ai and compute," May 2018. [Online]. Available: https://openai.com/blog/ai-and-compute/

[5] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

[6] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo, "Tiresias: A GPU cluster manager for distributed deep learning," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 485–500. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/gu

[7] 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 *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 463–479.

[8] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 583–598.

[9] P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations," *Journal of Parallel and Distributed Computing*, vol. 69, no. 2, pp. 117–124, 2009.

[10] Y. Li, J. Park, M. Alian, Y. Yuan, Z. Qu, P. Pan, R. Wang, A. Schwing, H. Esmaeilzadeh, and N. S. Kim, "A network-centric hardware/algorithm co-design to accelerate distributed training of deep neural networks," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 175–188.

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

[12] S. Wang, D. Li, J. Geng, Y. Gu, and Y. Cheng, "Impact of network topology on the performance of dml: Theoretical analysis and practical factors," in *IEEE INFOCOM 2019-IEEE conference on computer communications*. IEEE, 2019, pp. 1729–1737.

[13] Baidu. Allreduce. [Online]. Available: https://github.com/baidu-research/baidu-allreduce

[14] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," *The International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[15] Q. Cai, S. Chaudhary, M. Vuppalapati, J. Hwang, and R. Agarwal, "Understanding host network stack overheads," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021, pp. 65–77.

[16] J. Geng, D. Li, Y. Cheng, S. Wang, and J. Li, "Hips: Hierarchical parameter synchronization in large-scale distributed machine learning," in *Proceedings of the 2018 Workshop on Network Meets AI & ML*, 2018, pp. 1–7.

[17] M. Cho, U. Finkler, D. Kung, and H. Hunter, "Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy," *Proceedings of Machine Learning and Systems*, vol. 1, pp. 241–251, 2019.

[18] D. Xiao, X. Li, J. Zhou, Y. Du, and W. Wu, "Iteration number-based hierarchical gradient aggregation for distributed deep learning," *The Journal of Supercomputing*, vol. 78, no. 4, pp. 5565–5587, 2022.

[19] H. Mikami, H. Suganuma, Y. Tanaka, Y. Kageyama *et al.*, "Imagenet/resnet-50 training in 224 seconds," *arXiv preprint arXiv:1811.05233*, pp. 770–778, 2018.

[20] Z. Wang, H. Xu, J. Liu, H. Huang, C. Qiao, and Y. Zhao, "Resource-efficient federated learning with hierarchical aggregation in edge computing," in *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, 2021, pp. 1–10.

[21] Y. Jiang, H. Gu, Y. Lu, and X. Yu, "2d-hra: Two-dimensional hierarchical ring-based all-reduce algorithm in large-scale distributed machine learning," *IEEE Access*, vol. 8, pp. 183 488–183 494, 2020.

[22] Q. Luo, J. He, Y. Zhuo, and X. Qian, "Prague: High-performance heterogeneity-aware asynchronous decentralized training," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 401–416.

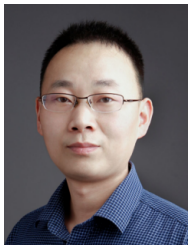
[23] Amazon elastic compute cloud (ec2). [Online]. Available: https://aws.amazon.com/ec2/

[24] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, "Analysis of {Large-Scale}{Multi-Tenant}{GPU} clusters for {DNN} training workloads," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 947–960.

[25] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *13th USENIX Symposium*

on Operating Systems Design and Implementation (OSDI 18), 2018, pp. 595–610.
[26] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, "Rdma over commodity ethernet at scale," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 202–215.
[27] A. Kashyap and X. Lu, "Nvme-oaf: Towards adaptive nvme-of for io-intensive workloads on hpc cloud," in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, 2022, pp. 56–70.
[28] Q. Ren, L. Zhou, Z. Xu, Y. Zhang, and L. Zhang, "Packetusher: Exploiting dpdk to accelerate compute-intensive packet processing," *Computer Communications*, vol. 161, pp. 324–333, 2020.
[29] ebpf documentation. [Online]. Available: https://ebpf.io/
[30] 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 *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 41–54.
[31] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, "Dcell: a scalable and fault-tolerant network structure for data centers," in *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, 2008, pp. 75–86.
[32] H. Abu-Libdeh, P. Costa, A. Rowstron, G. O'Shea, and A. Donnelly, "Symbiotic routing in future data centers," in *Proceedings of the ACM SIGCOMM 2010 conference*, 2010, pp. 51–62.
[33] Y. Tanaka and Y. Kageyama, "Imagenet/resnet-50 training in 224 seconds."
[34] C. Lao, Y. Le, K. Mahajan, Y. Chen, W. Wu, A. Akella, and M. Swift, "{ATP}: In-network aggregation for multi-tenant learning," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021, pp. 741–761.
[35] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. Ports, and P. Richtárik, "Scaling distributed machine learning with in-network aggregation," *arXiv preprint arXiv:1903.06701*, 2019.
[36] 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 *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021, pp. 676–691.
[37] M. Wu, Q. Chen, and J. Wang, "Bpcm: a flexible high-speed bypass parallel communication mechanism for gpu cluster," *IEEE Access*, vol. 8, pp. 103 256–103 272, 2020.
[38] Y. Ghigoff, J. Sopena, K. Lazri, A. Blin, and G. Muller, "{BMC}: Accelerating memcached using safe in-kernel caching and pre-stack processing," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021, pp. 487–501.
[39] Vgg19. [Online]. Available: https://pytorch.org/vision/stable/models/generated/torchvision.models.vgg19.html
[40] iftop. [Online]. Available: http://www.ex-parrot.com/%7Epdw/iftop/
[41] mpstat. [Online]. Available: http://sebastien.godard.pagesperso-orange.fr/man%5fmpstat.html
[42] E. Maniloff, S. Gareau, and M. Moyer, "400g and beyond: Coherent evolution to high-capacity inter data center links," in *2019 Optical Fiber Communications Conference and Exhibition (OFC)*. IEEE, 2019, pp. 1–3.
[43] W. Wang, D. Wu, S. Das, A. Rahbar, A. Chen, and T. E. Ng, "{RDC}:{Energy-Efficient} data center network congestion relief with topological reconfigurability at the edge," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 1267–1288.
[44] MELLANOX. Connectx-7 product brief. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/networking/ethernet-adapters/connectx-7-datasheet-Final.pdf
[45] L. Jia, R. Rajaraman, and T. Suel, "An efficient distributed algorithm for constructing small dominating sets," *Distributed Computing*, vol. 15, no. 4, pp. 193–205, 2002.
[46] S. McCanne and V. Jacobson, "The bsd packet filter: A new architecture for user-level packet capture." in *USENIX winter*, vol. 46, 1993.
[47] Cilium. [Online]. Available: https://github.com/cilium
[48] Bpf and xdp reference guide. [Online]. Available: https://docs.cilium.io/en/stable/bpf/
[49] K. K. Ram, J. R. Santos, Y. Turner, A. L. Cox, and S. Rixner, "Achieving 10 gb/s using safe and transparent network interface virtualization," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, 2009, pp. 61–70.
[50] O. Hohlfeld, J. Krude, J. H. Reelfs, J. Rüth, and K. Wehrle, "Demystifying the performance of xdp bpf," in *2019 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2019, pp. 208–212.
[51] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The express data path: Fast programmable packet processing in the operating system kernel," in *Proceedings of the 14th international conference on emerging networking experiments and technologies*, 2018, pp. 54–66.
[52] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.
[53] Byteps performance when training cnn. [Online]. Available: https://github.com/bytedance/byteps/blob/master/docs/performance.md
[54] dataloader. [Online]. Available: https://github.com/pytorch/pytorch/blob/master/torch/utils/data/dataloader.py#L195
[55] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.
[56] Z. Dai, X. Wang, P. Ni, Y. Li, G. Li, and X. Bai, "Named entity recognition using bert bilstm crf for chinese electronic health records," in *2019 12th international congress on image and signal processing, biomedical engineering and informatics (cisp-bmei)*. IEEE, 2019, pp. 1–5.
[57] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing, "Poseidon: An efficient communication architecture for distributed deep learning on {GPU} clusters," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 181–193.
[58] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo, "A generic communication scheduler for distributed dnn training acceleration," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 16–29.
[59] An instant virtual network on your laptop. [Online]. Available: http://mininet.org
[60] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *ACM SIGCOMM computer communication review*, vol. 38, no. 4, pp. 63–74, 2008.
[61] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella, "Presto: Edge-based load balancing for fast datacenter networks," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 465–478, 2015.
[62] V. Jalaparti, I. Bliznets, S. Kandula, B. Lucier, and I. Menache, "Dynamic pricing and traffic engineering for timely inter-datacenter transfers," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 73–86.
[63] D. S. Johnson, "Approximation algorithms for combinatorial problems," *Journal of computer and system sciences*, vol. 9, no. 3, pp. 256–278, 1974.
[64] L. Lovász, "On the ratio of optimal integral and fractional covers," *Discrete mathematics*, vol. 13, no. 4, pp. 383–390, 1975.
[65] 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*, vol. 26, no. 4, pp. 1837–1850, 2018.
[66] N. Matni and S. Tu, "A tutorial on concentration bounds for system identification," in *2019 IEEE 58th Conference on Decision and Control (CDC)*. IEEE, 2019, pp. 3741–3749.

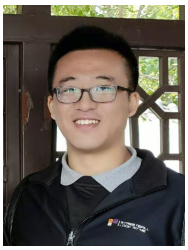
Peng Yang is currently pursuing the Eng.D. degree in computer science with the University of Science and Technology of China. His main research interests include eBPF/XDP technology, data center networks, and networking for AI.



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, China, in 2002 and 2007, respectively. He is currently a Professor with the School of Computer Science and Technology, University of Science and Technology of China (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 Thing. 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.

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 cloud computing, software-defined networking, data center networks, and networking for AI.

Qianyu Zhang is currently pursuing the Ph.D. degree in computer science with the University of Science and Technology of China. His main research interests are cloud/datacenter networks, eBPF/XDP technology, and networking for AI.



Jiawei Liu received the B.S. degree in the College of Computer Science and Technology, Jilin University in 2014. He is pursuing the Eng.D. degree in computer technology at the University of Science and Technology of China. His current research interests include software-defined networks, cloud computing and programmable networks.

Chunming Qiao (Fellow, IEEE) is a SUNY Distinguished Professor and also the current Chair of the Computer Science and Engineering Department at the University at Buffalo. He was elected as an IEEE Fellow for his contributions to optical and wireless network architectures and protocols. His current focus is on connected and autonomous vehicles, and quantum networks.