XAgg: Accelerating Heterogeneous Distributed Training Through XDP-Based Gradient Aggregation

Qianyu Zhang, Gongming Zhao, *Member, IEEE*, Hongli Xu, *Member, IEEE*, and Peng Yang

***Abstract*— With the growth of model/dataset/system size for distributed model training in datacenters, the widely used Parameter Server (PS) architecture suffers from communication bottleneck of gradient transmission. Recent works attempt to utilize programmable switches to implement in-network gradient aggregation and alleviate communication bottlenecks on PSs. Due to the limited on-chip memory of programmable switches, gradient transmission requires strict synchronization to achieve ideal aggregation performance. However, the distributed training system is usually heterogeneous in datacenters (*e.g.*, computation and bandwidth heterogeneity), and the gradient will reach the aggregation nodes asynchronously, thereby seriously affecting the aggregation performance. To solve the above issue, we propose XAgg, which accelerates heterogeneous gradient aggregation by deploying the eXpress Data Path (XDP) based aggregator on servers. Specifically, the abundant idle memory on servers can cache the entire gradient, so as to effectively deal with asynchronous gradient transmission in heterogeneous scenarios. Moreover, XDP can provide high-performance and low-latency gradient aggregation. We conduct microbenchmark and testbed with real-world DNN models and datasets. Experimental results show that XAgg improves the gradient aggregation throughput by 3.3× compared with TCP-based aggregation, reaching 100 Gbps with 10 CPU cores. In addition, XAgg reduces communication time by 49%-82% compared with state-of-the-art solutions.**

***Index Terms*— In-network gradient aggregation, distributed model training, eXpress data path (XDP).**

I. Introduction

IN RECENT years, the development of Deep Neural Network (DNN) has led to breakthroughs in various fields, including computer vision [1], natural language processing [2] and recommendation systems [3]. With the increasing complexity of applications, the current DNN training shows the trend of building increasingly sophisticated models on large datasets [4] for better prediction performance. In order to cope with the surge of training computation, distributed training (DT) is widely deployed in datacenters, and usually includes two types of nodes: workers and parameter servers (PSs) [5]. Specifically, in each iteration, the workers first train the models and transmit their local gradients to the PS(s), where gradients are tensors, *i.e.*, arrays of values. PS(s) are responsible for aggregating the gradients from all workers and sending the aggregated gradients back to workers. Typically, training a DNN model requires hundreds of iterations on the dataset until convergence [6], which is a time-consuming process.

With the development of GPU [7] and other hardware accelerators [8], [9], computation performance has been improved 62× over the past 7 years [10], [11]. As a result, the performance bottleneck of DT in datacenters has gradually shifted from computation to communication, *i.e.*, gradient transmission [10], [12]. For example, when a PS-based DT task trains the DeepLight model with 100 Gbps bandwidth, 79% of the total time is occupied for intra-cluster communication [13]. Prior works try to alleviate the communication bottleneck through gradient compression [12], [14], [15], [16] or communication scheduling [17], [18], [19], [20], [21]. For example, OmniReduce [12] leverages gradient sparsity and only sends non-zero data blocks to minimize traffic transfer. However, gradient compression suffers from a decrease of training accuracy. As the representative of communication scheduling, BytePS [21] transmits gradients in layers of the model, increasing the overlap between local training and gradient transmission, but does not actually reduce the total transmission amount. Nowadays, with the increasing computation capacity provided by programmable switches, some works propose in-network aggregation (INA) as a promising solution [13], [22]. Specifically, INA utilizes programmable switches to aggregate gradients in the transmission path from workers to PS(s), and these programmable switches only send the aggregated results to PS(s). In this way, INA helps to reduce the transmission traffic from workers to PS(s), improve training throughput and accelerate the model training.

However, the current in-network aggregation methods based on P4 programmable switches suffer from limited on-chip memory, and the aggregation performance will be seriously affected especially in heterogeneous scenarios. On the one hand, there is a gap between the on-chip memory size of programmable switches (*e.g.*, the on-chip memory of Wedge100BF-32x is 22MB [23]) and the gradient size (*e.g.*, the size of BERT model is 1274MB [13]). Existing solutions divide the local gradients of workers into fixed size fragments,

Manuscript received 25 February 2023; revised 7 August 2023; accepted 30 November 2023; approved by IEEE/ACM Transactions on Networking Editor J. P. Jue. This work was supported in part by the National Science Foundation of China (NSFC) under Grant 62372426 and Grant 62102392, in part by the Open Research of Projects of Zhejiang Laboratory under Grant 2022QA0AB04, in part by the National Science Foundation of Jiangsu Province under Grant BK20210121, in part by the Hefei Municipal Natural Science Foundation under Grant 2022013, in part by the Youth Innovation Promotion Association of Chinese Academy of Sciences under Grant 2023481, and in part by the Fundamental Research Funds for the Central Universities. *(Corresponding authors: Gongming Zhao; Hongli Xu.)*

The authors are with the School of Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui 230027, China, and also with the Suzhou Institute for Advanced Research, University of Science and Technology of China, Suzhou, Jiangsu 215123, China (e-mail: zqy2019@mail.ustc.edu.cn; gmzhao@ustc.edu.cn; xuhongli@ustc.edu.cn; ypholic@mail.ustc.edu.cn).

Digital Object Identifier 10.1109/TNET.2023.3339524

and partition the memory of the programmable switch into units of the same size [13], [22]. The gradients of all workers should arrive strictly synchronously to obtain the optimal aggregation rate, as a memory unit can only store gradient fragments with the same index. On receiving the same gradient fragments from all workers, the switch completes the aggregation of this fragment, and the corresponding memory unit can be reused for subsequent aggregation. On the other hand, the practical system is usually heterogeneous, including heterogeneous computation capacity and link bandwidth. However, it is difficult for the heterogeneous system to achieve strict synchronization of gradient transmission, because the actual aggregation performance of programmable switches is affected by the short board effect. Specifically, the worker(s) with the weakest computation capacity determine the local training completion time, and the minimum link bandwidth limits the aggregation rate.

An intuitive solution is expanding the on-chip memory of the programmable switch so as to store the entire model. But this means complex chip reconfiguration and huge costs. Alternatively, we observe that there are abundant idle CPU and memory resources on the servers in the datacenter. For example, Alibaba's statistics show that there are 60% idle CPUs and 35% free memory in its datacenters [24], [25]. This is because the load balancer uses a conservative resource allocation strategy to reserve massive CPU/memory resources to cope with performance spike [25]. We explore deploying gradient aggregation programs on one or more servers before the PS, utilizing the remaining computation and memory resources to accelerate gradient aggregation. The abundant idle memory on servers can cache the entire gradient and wait for stragglers to catch up, so as to deal with computation/bandwidth heterogeneity.

However, two factors will limit the performance of gradient aggregation with idle resources on servers. First, the performance bottleneck of the kernel network stack makes it difficult to achieve high throughput gradient aggregation and forwarding on servers [26], [27]. Second, gradient traffic may be aggregated and forwarded across several servers, resulting in high transmission delay, even up to hundreds of milliseconds [28].

Fortunately, eXpress Data Path (XDP), as a pre-stack packet acceleration technology [29], [30], can effectively eliminate the performance bottleneck from the network stack. Specifically, XDP offloads gradient aggregation logic to the NIC driver in front of the network stack. The significant advantage of XDP is that it can achieve high performance and low latency gradient aggregation at 100Gbps. Meanwhile, the throughput of TCP-based aggregation only reaches 30 Gbps by the experiments in §VI-B. In addition, XDP can organize idle memory resources through the eBPF map [30], and caches the gradients of the entire model on hosts, so as to guarantee the aggregation performance in heterogeneous scenarios. Moreover, XDP has additional benefits, including flexible deployment without hardware support, less impact on non-gradient traffic transmission and on-demand CPU resources occupation.

In this paper, we propose a novel distributed training framework, called XAgg, which deploys XDP-based aggregators on servers in datacenters to accelerate model training of heterogeneous systems. The XDP-based aggregator can realize gradient aggregation and forwarding at 100 Gbps line rate with low latency. Although XDP has effectively improved the aggregation performance of a single host, XAgg still faces two challenges in building an reliable and efficient aggregation tree. First, the existing ACK-based reliable transmission mechanism [13], [22] is complex to implement, and brings high overhead, which significantly degrades the aggregation performance of XAgg. Second, the existing aggregation tree construction strategies based on multicast [31] or Steiner tree [32] usually need to solve linear programming, and their high time complexity is not suitable for XAgg.

The main contributions of this paper are as follows:

- We propose XAgg, which deploys XDP-based aggregators on servers to achieve high-performance and low-latency gradient aggregation, thus accelerating heterogeneous distributed training.
- XAgg adopts a group-based acknowledgment mechanism for reliable transmission.
- XAgg proposes a low-time complexity strategy based on k-tree to construct the aggregation tree so as to effectively utilize the idle resources on servers. Moreover, the aggregation topology can be dynamically adjusted according to the server load.
- We conduct testbed with real-world DNN models and datasets. The experimental results show that XAgg improves the aggregation throughput by 3.3× compared with TCP-based aggregation. In addition, XAgg reduces communication time by 49%-82% compared with state-of-the-art solutions.

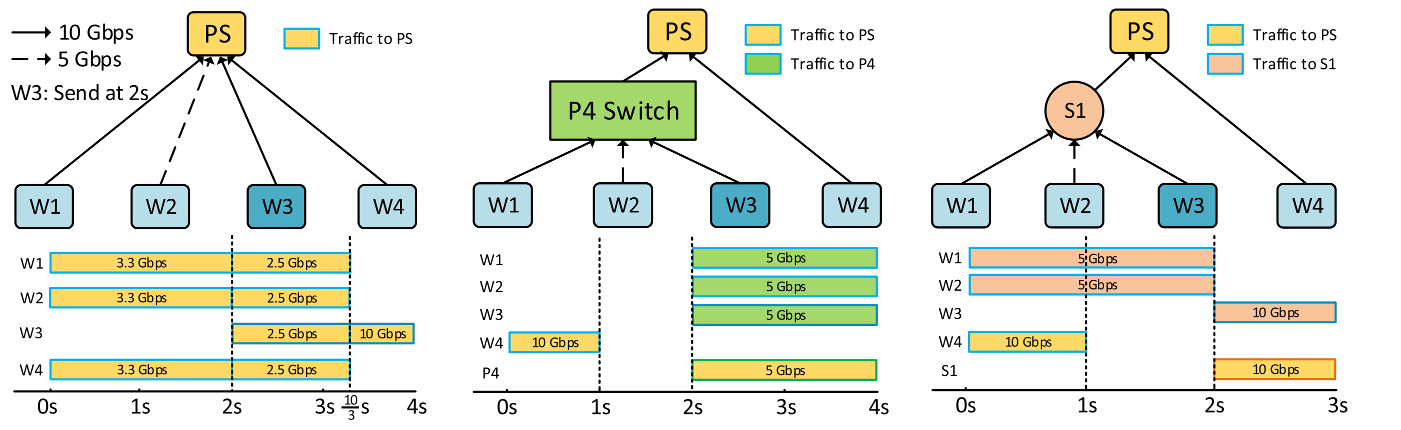
II. Background and Motivation

We give an example to analyze the pros and cons of two existing solutions for distributed model training and motivate our study in this section.

A. A Motivation Example

Consider a distributed model training system containing 1 PS and 4 workers (*i.e.*, W_1, W_2, W_3 and W_4). To simulate bandwidth heterogeneity, the ingress/egress bandwidth of PS, W_1, W_3 and W_4 is set to 10 Gbps, while that of W_2 is set to 5 Gbps. To simulate computation heterogeneity, each iteration of local model training for W_3 is set to complete two seconds slower than other three workers. In addition, for the convenience of calculation, we assume that the gradient size of the model is 1.25 GB. Thus, it takes 1 second for a worker to send the local gradient to the PS with 10 Gbps bandwidth.

As a classic framework, Parameter Server (PS) is widely adopted in distributed model training [5], [33]. As shown in Fig. 1(a), after a iteration of training, workers push their local gradients to the PS over the network for aggregation. Then, workers pull the updated gradients from the PS for the next iteration of training. In our example, when W_1, W_2 and W_4 finish training, they share the 10 Gbps bandwidth of PS and send gradients at $\frac{10}{3}$ Gbps in 0-2s. Then,



(a) Parameter Server (PS) (b) Programmable Switch (c) Our Intuition: Using Idle Servers

Fig. 1. A distributed training task contains 1 PS and 4 workers, each equipped with a 10 Gbps NIC. The available ingress/egress bandwidth of PS, W_1, W_3 and W_4 is 10 Gbps, while that of W_2 is 5 Gbps. Moreover, each iteration of training for W_3 completes two seconds later than the other three workers. (a) PS model takes 4s to complete aggregation. (b) Heterogeneous link bandwidth and computation capacity seriously affect the aggregation performance of programmable switches, and its actual aggregation time is also 4s. (c) The XDP-based aggregation program is deployed on servers in the datacenter, using idle CPU and memory for gradient pre-aggregation and caching, respectively. The aggregation time is only 3s.

W_3 completes training and starts sending gradients at 2s, and the transmission rate of the four workers reduce to 2.5 Gbps. W_1, W_2 and W_4 complete the gradient transmission at $\frac{10}{3}s$. Meanwhile, W_3 monopolizes all the bandwidth, increasing transmission rate to 10 Gbps. Finally, W_3 completes the gradient transmission at 4s. Since all workers need to share the network bandwidth of PS, gradient transmission will bring a communication bottleneck to distributed model training. Even if we adopt pipeline transmission, the bandwidth bottleneck of PS determines that the theoretical communication time is still at least 4s.

To reduce gradient transmission overhead from workers to the PS and accelerate training, several prior works realize in-network aggregation with programmable switches, such as SwitchML [13] and ATP [22]. In an ideal situation, all workers start sending gradients at the same rate simultaneously. The programmable switch can efficiently aggregate the received gradients and forward the aggregated results to the PS, which can significantly reduce the gradient traffic sent to the PS and eliminate the bandwidth bottleneck. In practice, workers cannot guarantee strict synchronization of gradient transmission, that is, there are inconsistencies in the gradient transmission bandwidth and start time among different workers. Because the limited on-chip memory of the programmable switch cannot cache all unaggregated gradients, workers will not send the subsequent gradients until receiving the previous aggregated results. Therefore, the actual aggregation bandwidth of the programmable switch depends on the smallest transmission rate, and the aggregation start time depends on the latest training completion time.

Fig. 1(b) illustrates how heterogeneous bandwidth and computation capacity slow down the aggregation time of the programmable switch. When W_1, W_2 and W_4 finish training and start to send gradients, the memory of the programmable switch is quickly filled with the gradient fragments of W_1 and W_2. Because W_3 has not finished training, the programmable switch cannot complete the gradient aggregation and has to wait. Meanwhile, W_4 sends its gradients to the PS at 10 Gbps in 0-1s. Then, W_3 completes training and starts sending gradients at 2s, and the programmable switch starts the aggregation operation and pushes the aggregated gradients to the PS. Programmable switch cannot perform in-network aggregation at full bandwidth, because the transmission bandwidth of W_2 is only 5 Gbps. As a result, the entire aggregation process takes 4s, and the programmable switch does not fully utilize its processing capability to accelerate gradient aggregation.

B. Our Intuition

We observe from the above examples that the in-network aggregation performance will be significantly affected if the gradient transmission of the workers is not synchronized. One possible solution is to add more on-chip SRAM memory or external on-host DRAM memory to programmable switches. However, adding on-chip SRAM memory is impractical due to cost and complexity concerns. Meanwhile, using the DRAM of the directly connected server for storage expansion has the disadvantage of insufficient bandwidth between the switch and the server. Therefore, the memory of programmable switch will not increase dramatically in the future.

The underutilized CPU and memory resources of servers in the datacenter inspire our solution. Statistics show that the remaining memory on servers is usually abundant, even up to tens of GB [24], [25]. We can deploy gradient aggregators on servers and use the remaining bandwidth, computation and memory resources to undertake in-network aggregation. On the one hand, idle servers expand the aggregation bandwidth. On the other hand, the abundant idle memory can cache the whole model. Thus, we explore the potential of gradient aggregation using idle resources in accelerating asynchronous gradient transmission.

Fig. 1(c) shows that gradient pre-aggregation with idle resources on servers can fully utilize bandwidth and reduce transmission time. When W_1, W_2 and W_4 finish training, they all send gradients to their aggregation nodes. Specifically, W_1 and W_2 send gradients to server S_1 at 5 Gbps in 0-2s, and W_4 takes one second to send gradients to the PS at 10 Gbps. Then, W_3 finishes training and starts sending gradients at 2s.

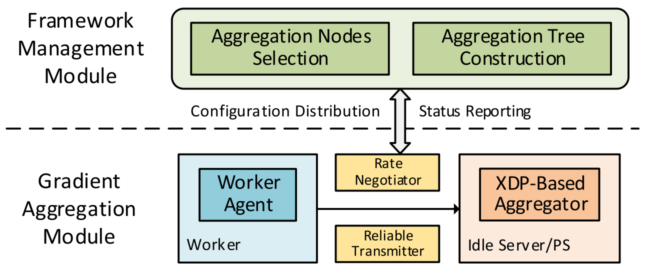

Framework Management Module
Aggregation Nodes Selection
Aggregation Tree Construction
Configuration Distribution
Status Reporting
Gradient Aggregation Module
Worker Agent
Worker
Rate Negotiator
Reliable Transmitter
XDP-Based Aggregator
Idle Server/PS

Fig. 2. XAgg System Overview. Gradient Aggregation Module deploys XDP-based aggregators on servers to accelerate gradient aggregation. Framework Management Module is responsible for selecting aggregation nodes from servers and constructing the aggregation tree.

As S_1 has aggregated and stored the gradients of W_1 and W_2, it can immediately aggregate the received gradients from W_3 and send the aggregated results to the PS.

C. Challenges

Line Rate Aggregation. The main challenge for gradient aggregation on servers is to implement gradient aggregation and forwarding at line rate. Many prior works [26], [27], [34] have shown that the kernel network stack is the main bottleneck of data transmission in datacenters. In addition, massive traffic will significantly increase the processing delay of the network stack. For example, concurrent TCP connections will result in long latency due to queueing delays, even up to hundreds of μs [27]. As a result, massive gradient aggregation and forwarding will cause huge overhead and delay due to the complex processing of the network stack.

DPDK vs. XDP. We seek a network acceleration technology for high-performance gradient aggregation and forwarding. We first consider the widely used framework DPDK [35], which is a kernel bypass approach for high-speed packet processing. Although DPDK offers the highest performance among the existing frameworks [36], it is not suitable to deploy on servers for gradient aggregation. DPDK needs to bind a dedicated NIC and moves the NIC's control from the kernel to the user space. This will bring management issues to the server and affect the transmission of regular traffic, especially in the case that cheap/sparse CPU machines are not always equipped with dedicated NICs. Second, DPDK needs to bind dedicated CPU cores and adopts busy polling to process packets. Thus, the usage of these dedicated CPU cores is always pegged at 100%, even without any packets, which brings large unnecessary overhead to servers.

XDP is a pre-stack packet acceleration technology that has been widely used in load balancing, DDoS detection and other fields [29], [30]. Unlike DPDK, XDP processes gradient packets before the network stack, and other traffic will still be processed by the stack. Thus, XDP will not affect the regular traffic of servers, and can use the management interface and security guarantees offered by kernel [29]. In addition, XDP requires CPU resources on demand instead of occupying dedicated CPU cores, which can effectively save the power of selected servers. Finally, we choose XDP as the data plane technology for the gradient aggregation on servers, which can achieve high-performance and low-latency gradient aggregation with less overhead.

III. System Overview

We propose an XDP-based gradient aggregation framework, called XAgg. XAgg leverages the underutilized CPU and memory resources of servers in datacenters for gradient aggregation, thereby accelerating distributed model training, especially with heterogeneous bandwidth and device performance. As shown in Fig. 2, XAgg consists of two core modules: Gradient Aggregation and Framework Management.

Gradient Aggregation Module deploys XDP-based aggregators on servers to provide high-performance and low-latency gradient aggregation services for distributed model training. The selected servers, workers and the PS form a gradient aggregation tree. In the aggregation tree, the gradient sender is regarded as the child node, and the gradient receiver is regarded as the parent node, *i.e.*, the aggregation node. Specifically, the gradient aggregation module contains four components as follows:

- **Worker Agent.** When the worker completes a iteration of local training, the worker agent will encapsulate the gradient fragments into the UDP packets with the customized format, and send them to the aggregation node (§IV-A).
- **XDP-based Aggregator.** XAgg deploys an XDP aggregation program on each selected servers and the PS. When the aggregator receives the gradient packets from the child node, it will perform the aggregation operation and send the aggregated results to its parent node, *i.e.*, upper aggregator (§IV-B).
- **Rate Negotiator.** XAgg performs rate negotiation between parent and child nodes to fully utilize bandwidth while avoiding congestion. Rate negotiation includes initialization and update periods (§IV-C).
- **Reliable Transmitter.** XAgg adopts group-based acknowledgement mechanism to achieve reliable transmission with less overhead (§IV-D).

Framework Management Module manages the entire XAgg distributed model training system. It is responsible for selecting aggregation nodes from servers and constructing the aggregation tree (§V). In addition, the aggregation topology can be dynamically adjusted according to the load status reported by servers.

IV. Aggregation Module Design

A. Worker Agent Design

XAgg Packet Format. XAgg transmits gradient fragments based on IP protocol. Specifically, XAgg customizes the payload of IP packet, which contains the XAgg header field and the gradient array field. Fig. 3 shows the packet format of an XAgg gradient fragment. The `XAgg Header` field contains metadata about the fragment. The `Host ID` field is a 32 bits one-hot encoding and is used to identify each host (worker, aggregation server and PS) in the network. The `Gradient ID` field is the identifier of a gradient data fragment. The `Gradient Array` field contains 256×4 bytes gradient values.

Gradient Array Size. The size of the gradient array transmitted by each XAgg packet needs to be carefully

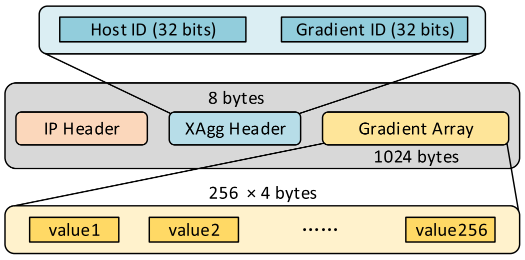


Fig. 3. XAgg Packet Format. It customizes the payload of IP packet, including the XAgg header and the gradient array field.

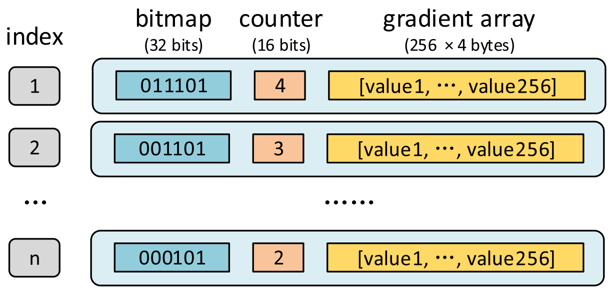


Fig. 4. Aggregator Storage Format. XAgg on each server caches the received gradients by an array map. Each array element stores the aggregated gradients with the same index.

determined. Obviously, a packet carrying more gradient data would improve the goodput and reduce the overhead of packet header processing. For example, the total Ethernet frame length of a 320-element gradient payload is 1328 bytes, and its goodput is 96.39%. While the goodput of a 32-element payload is only 72.73%. However, an excessively large gradient array size will increase the aggregation time of one packet, which could lead to the loss of later arriving packets, thereby reducing the actual gradient throughput. Therefore, there is a trade-off between theoretical goodput and actual gradient throughput. In our implementation, we select 256-element gradient payload according to the experimental results in §VI-B.

Floating-Point Numbers. Gradient values are usually floating point type. However, the current XDP program does not support floating point operations and only supports integers. We adopt a scheme similar to the prior works [13], [22], where the workers multiply the floating point number by a factor (*i.e.*, 10^8) and then round it to a 32-bit integer. The aggregators only perform integer addition operations. When workers receive the aggregated gradient data broadcast by PS, they convert these 32-bit integers back to floating point numbers through dividing by the factor, and then divide by the number of workers.

Agent Interacts with ML Framework. The interaction between the worker agent and the ML framework is mainly in gradient sending and receiving phases. XAgg uses PyTorch [37] and can be easily ported to other ML frameworks. After a round of training, the agent converts the gradient into a one-dimensional vector. In order to fully utilize bandwidth, the agent uses multiple processes to send gradients in parallel. The agent divides the vector into several parts, each of which is delivered to the C-based sending process and sent to the aggregation node. The aggregated gradient will be received by XDP deployed on the worker side and cached in the eBPF map, which is similar to the implementation of aggregation nodes. The worker agent will then obtain the gradient vector from the eBPF map and convert it to the model parameter format, passing it to Pytorch for the next round of training.

Algorithm 1 XDP-Based Aggregator Logic

```
Initialize Aggregator Metadata:
1:  Achieve child and parent node(s) from the control plane.
2:  n = number of child node(s).
3:  N = number of gradient fragments.
4:  Agg_array[0:N] = {0}.
Upon Receiving Gradient pkt{id, idx, grad[0:256]}:
5:  agg = Agg_array[pkt.idx]
6:  if agg.bitmap == (agg.bitmap | pkt.id) then
7:      Drop pkt.
8:  agg.bitmap += pkt.id
9:  agg.counter++
10: for $i$ in [0:256] do
11:     agg.grad[i] += pkt.grad[i]
12: if agg.counter == n then
13:     Forward agg to parent node.
After an Epoch, Reset Aggregator Metadata.
```

B. XDP-Based Aggregator Design

Aggregator Storage Format. XDP-based aggregator on each server caches the received gradient data by an array map [30]. As shown in Fig. 4, each array element stores the aggregated gradient data of XAgg packets with the same index and contains several fields. The `bitmap` field records which child node's gradient fragment has been received and aggregated by the aggregator. The `counter` field indicates the number of child nodes aggregated in the gradient data. The `gradient array` field stores aggregated gradient from distinct child nodes and contains 256 gradient values.

Aggregator Workflow. The workflow of XDP-based aggregator on each server is describe as Alg. 1. The aggregator first initializes its metadata, including child/parent node(s), number of gradient fragments and aggregator array. Once a gradient packet `pkt` arrives, the aggregator first locates the particular array element `agg` by `pkt.idx`, and checks whether `agg` contains the gradient data of `pkt` or not. If not, the aggregator updates the `agg.bitmap` and increases the `agg.counter` by one. The aggregator then adds the gradient values of `pkt` and the local gradient array `agg.grad` in order. If the `agg.counter` equals the number of child nodes N, the aggregation of the gradient fragment is completed, and the aggregator will forward this gradient fragment to its parent node. When the aggregation of an epoch completes, the aggregator will reset the metadata.

C. Rate Negotiator

XAgg wants to make full use of the available bandwidth of the link between servers while avoiding loss of gradient packets caused by congestion. Each node is equipped with a rate negotiator, which dynamically adjusts the gradient transmission rate according to the real-time remaining bandwidth of the link. Rate negotiation is divided into rate initialization and rate update periods.

Rate Initialization. When a worker completes local training, it will request the initial gradient sending rate from its parent node. We use $P = \{p_1, p_2, \ldots, p_{|P|}\}$ to denote the parent nodes, *i.e.*, aggregators, including the selected servers and the PS. We denote $C_p = \{c_1, c_2, \ldots, c_{n_p}\}$ as the child nodes set of parent node $p \in P$, where n_p is the number of child nodes of the parent node p. Let $b^{tx}_{c,p}$ represent the sending bandwidth from the child node $c \in C_p$ to its parent node $p \in P$. In addition, we use B^{in}_p and B^{out}_p to denote the idle ingress and egress bandwidth available to the parent node p, respectively. Therefore, for each parent node $p \in P$, its ingress and egress bandwidth should satisfy the following inequalities at every moment:

$$\begin{cases} \sum\limits_{c \in C_p} b^{tx}_{c,p} \le B^{in}_p, & \forall p \in P \\ \min\limits_{c \in C_p} b^{tx}_{c,p} \le B^{out}_p, & \forall p \in P \end{cases} \tag{1}$$

The first set of inequalities indicates that the total sending bandwidth of the child nodes should not exceed the maximum ingress bandwidth of their parent node. The second set of inequalities indicates that the minimum sending bandwidth among child nodes should not exceed the egress bandwidth of the parent node. This is because the aggregate gradient bandwidth of the parent node depends on the minimum sending rate among its child nodes.

In order to ensure that each aggregation node in XAgg satisfies the above bandwidth constraints, all workers will negotiate with their parent nodes (*i.e.*, aggregators) to obtain the initial gradient sending rate. Specifically, when a worker completes local model training, it first sends a ready notification to its parent node, and informs the parent node of its maximum sending bandwidth. After the parent node receives the gradient sending request from a worker, the request will be transferred to the upper layer in turn until the root node (*i.e.*, PS). When the PS receives the request, it will calculate the sending bandwidth for its child nodes according to Iq. (1) and reply to the child node. After obtaining the calculation result, the child node of the PS will repeat the above calculation and reply operations. Finally, the worker starts sending the local gradient to its parent node (*i.e.*, aggregator) at the negotiated rate.

Rate Update. When a worker starts or finishes sending, the sending rate of other workers with the same parent node will be updated. First, when a worker completes training, it will request sending bandwidth from the parent node, and the total bandwidth will be reallocated. Other workers will be notified to reduce the sending rate to free up bandwidth for the newly joined worker. Second, when a worker finishes sending, it will notify its parent node. The parent node will reallocate the bandwidth released by this worker to other workers.

In addition, due to network dynamics, XAgg can adjust the sending rate of the workers to adapt to the real-time network status, so as to avoid network congestion and massive packet loss. Specifically, the aggregation node will count the packet loss rate for each worker. If the packet loss of one worker

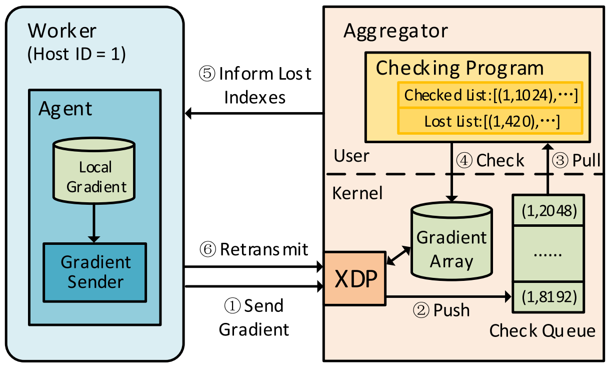


Fig. 5. Illustration of Group-Based Reliable Transmission. XAgg logically divides every 1024 packets into a group. When the aggregator receives the last packet of each group, it will check whether the packet of the group is received completely, and inform the corresponding worker of the lost indexes for retransmission.

exceeds the threshold (such as 1%) in a statistical period, the aggregator will notify the worker to reduce the sending bandwidth according to the packet loss rate. We will elaborate on how to count packet loss in the next section (§IV-D).

Rate Update Delay. The delay of rate initialization/update is very small. Experimental results show that the average delay of rate initialization and rate update is 9.6ms and 1.7ms, respectively, which is negligible compared with the training/gradient transmission time.

There is still a small probability that one worker completes training or finishes sending during the rate negotiation period. This worker will immediately notify its parent node, and its parent node will reallocate bandwidth and notify all affected child nodes. The send rate of the affected child nodes will adjust according to the latest received notification to deal with rate staleness.

D. Reliable Transmitter

A native solution for reliable transmission is to leverage the ACK acknowledgement mechanism like TCP protocol, which is adopted by some prior works [13], [22]. Considering that XAgg will perform rate negotiation (§IV-C) and aggregation nodes selection (§V), the sending rate of workers and aggregation topology will be determined according to the idle bandwidth and capacity of the aggregators. Therefore, the packet loss caused by network congestion and aggregator overload will be effectively reduced. On the other hand, replying ACK for each packet will bring extra overhead to both the aggregators and workers. Furthermore, it is too complicated to implement the delayed acknowledgment in the XDP program, and the overhead will degrade the aggregation performance.

As a result, XAgg adopts group-based acknowledgement mechanism to achieve reliable transmission, where lost gradient packets are detected by receivers. Specifically, XAgg logically divides the gradient packets of each worker into groups by a configured number (*e.g.*, 1024). Once receiving the end packet of each group, the aggregator will lookup the corresponding index range in gradient array and check whether all packets of the group are received. If packet loss is found, the lost indexes will be fed back to the corresponding worker, and this worker will retransmit the lost gradient packets.

Fig. 5 illustrates the interaction between the worker (assuming the host id is 1) and the aggregator for reliable transmission, where the aggregator contains an XDP-based aggregation program and a checking program. The aggregator maintains an FIFO map (*i.e.*, *Check Queue*) in kernel space to store the indexes to be checked. In addition, the aggregator maintains a *Checked List* and a *Lost List* in user space. Upon receiving the gradient packets from worker, the aggregator calculates whether the packet is the last one of a group or not. If so, the aggregator will put this packet's host id and index into the *Check Queue*. XAgg sets 1024 packets as a group, which is beneficial to adopt bit operations to speed up the modulo calculation for judging the end packet. As shown in Fig. 5, the gradient packet with index 8192 is identified as the last one of a group, the tuple (1, 8192) will be pushed into the *Check Queue*.

Meanwhile, the aggregator pulls the index to be checked (*i.e.*, 2048) from the *Check Queue*, checks whether there is any packet loss in the [1025, 2048] interval of the gradient array, and appends tuple (1, 2048) to the *Checked List*. The aggregator then stores the indexes of the lost packets in the *Lost List* and informs the corresponding worker of the lost indexes. Next, the worker agent will retransmit the lost gradient packets.

Finally, when the worker completes the gradient sending, it will notify the aggregator agent. The aggregator performs the following three check items to ensure the correct reception of all gradient data. (i) Check the tail fragments that cannot be divided by 1024. (ii) Traverse the *Checked List* to confirm whether the end packet of each group has been checked or not, so as to avoid that the group is not checked due to the loss of the end packet. (iii) Traverse the *Lost List* to check whether all retransmitted packets have been received or not.

V. Management Module Design

XAgg proposes a k-tree based aggregation topology construction strategy to maximize the utilization of idle computation and bandwidth resources of servers. The aggregation topology construction includes three steps: worker performance evaluation, aggregation nodes selection and aggregation tree construction.

We note that the selection of aggregation nodes can be transformed into a multicast problem [31] or a terminal Steiner tree problem [32], that is, constructing multicast trees or selecting Steiner points. However, the above problems have been proved to be NP-hard [32], [38]. The existing methods usually solve linear programming to obtain approximate solutions [38], [39], [40], which takes a long time and is not practical enough. We propose a lightweight aggregation tree construction method based on k-ary tree.

We consider the widely used leaf-spine topology in datacenters, which can provide non-blocking connections for a large number of servers [41]. On the one hand, the link delay is predictable since communication between two servers in a two-layer leaf-spine topology goes through up to three switches. On the other hand, the link delay of the current datacenter has reduced to several tens of microseconds. For example, the TCP round-trip delay in Google datacenter is $\sim 40\mu$s [42]. According to Section VI-C, XAgg can reduce the per-epoch communication time of large size models by several seconds. Obviously, the benefit of aggregation acceleration is much greater than the cost of link delay.

Worker performance evaluation. XAgg first evaluates the overall performance of each worker, and workers with similar performance will be preferentially assigned to the same aggregation node. Specifically, we adopt the PS architecture for the first round of training, in which the training time t_1 and the transmission time t_2 of each worker is recorded [43]. In this way, we obtain the total time $t = t_1 + t_2$ of each round for each worker. We then sort t and build an aggregation tree. Therefore, the overall performance (training and communication) of the worker assigned to each aggregation node is similar. It should be noted that if the proportion of communication time is lower than the threshold (*e.g.*, 10%), the model does not need to be accelerated and XAgg will exit.

Aggregation Nodes Selection. We then select a set of alternative servers according to the resource usage reported by the servers. Specifically, we consider two screening criteria. (1) The sum of used memory and model size cannot exceed 80% of the total memory, so that there is redundant memory for new tasks on the server. (2) The idle bandwidth does not exceed $10n$ Gbps, where n is the number of idle CPU cores. Considering that a single CPU core can aggregate and forward gradients at a rate of $\sim$10 Gbps (§VI-B). In order to make full use of the bandwidth resources of the server, we hope that the gradient processing capacity of the idle CPU cores on the server can reach the available bandwidth. Then, we sort the idle resources of the alternative servers in descending order with the priority of {bandwidth, CPU, memory}.

Aggregation Tree Construction. Next, we construct a k-ary aggregation tree according to the number of workers, where each aggregation node has at most k child nodes. We use w to denote the number of workers. Let h denote the height of the aggregation tree. The k-ary aggregation tree needs to satisfy the following inequality:

$$\begin{cases} k^{h-2} < w \leq k^{h-1} \\ k, h, w \geq 2 \\ k, h, w \in N^* \end{cases} \tag{2}$$

We calculate $h = \lceil 1 + \log_k w \rceil$. In practice, we need to make a trade-off between the aggregation rate and the number of aggregation nodes, that is, to determine the number of child nodes k of the aggregation node. To simplify the analysis, we assume that each aggregation node's ingress and egress bandwidth is B. If k equals 2, the aggregation tree is a binary tree, and the theoretical aggregation bandwidth can reach half of the bandwidth of the aggregation nodes, but it requires more aggregation nodes. If k increases to a large value, the aggregation tree requires fewer aggregation nodes, but the theoretical aggregation bandwidth can only reach $\frac{1}{k}$ of the bandwidth B. In practice, we usually choose the value of k from 2 to 5 according to the number of alternative aggregation nodes.

We start building the aggregate tree from the bottom (*i.e.*, from worker side). We determine the number of child nodes

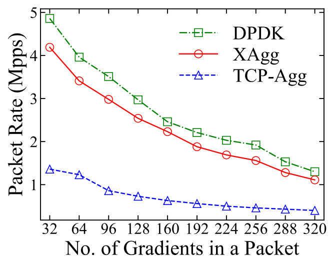


Fig. 6. Packet processing rate vs. No. of gradients in a packet.

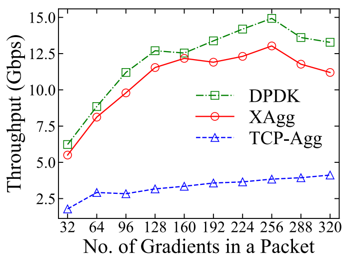


Fig. 7. Throughput vs. No. of gradients in a packet.

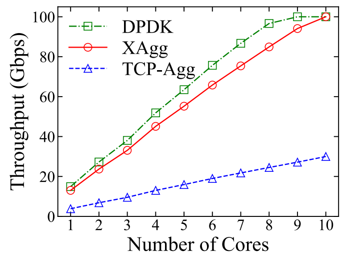


Fig. 8. Throughput vs. No. of CPU cores.

($\leq$k) that each candidate server can serve according to its available bandwidth. First, we take the first server from the candidate servers and assign it to k workers. Let's denote the ingress bandwidth of the selected server as B_1, then the average sending rate of the above k workers is $\frac{B_1}{k}$, and we denote this sending rate as the reference rate b. We continue to allocate idle servers to other workers. For server i, we calculate the value of $\lceil \frac{B_i}{b} - \frac{1}{2} \rceil$ as the number of its child nodes. We use the above method to further construct the aggregation tree layer by layer, and the root node of the aggregation tree is the PS.

After we finish constructing the aggregation tree, we will traverse all the aggregation nodes. If an aggregate node has only one child node, this node will be removed and its child node will be directly connected to its parent node. This is because the aggregation node with one child node cannot speed up gradient aggregation, but instead increases transmission delay and resource overhead. Accordingly, the total number of aggregation nodes is reduced by one. After traversal, we can obtain the required number of aggregation nodes and denote it as N. Finally, we select the top N as aggregation nodes from the sorted candidate servers and deploy the XDP-based aggregator on them.

Discussion. Considering the dynamic load on servers, the management module of XAgg will traverse the load state of servers after each iteration of aggregation and determine whether to adjust the aggregation topology. In addition, when a selected aggregation server encounters a burst load during an iteration of aggregation and cannot continue to undertake gradient aggregation. The management module will remove this server from the aggregation topology, and its child nodes will directly point to its parent node. Then, these child nodes will resend the entire gradient to the new parent node.

VI. Evaluation

In this section, we first give the metrics, benchmarks and testbed setups for performance evaluation of XAgg. Then, we conduct microbenchmark experiments to measure the performance advantages of XDP-based aggregation in terms of throughput, CPU usage and latency. Finally, we evaluate the efficiency of gradient aggregation of XAgg in heterogeneous scenarios compared to other benchmarks on the testbed.

A. Experimental Settings

Performance Metrics. We give an outline of the following two sets of performance metrics.

1) To illustrate the advantage of XDP-based aggregation, we adopt the following metrics in microbenchmark: (i) the *packet processing rate* (Mpps); (ii) the *gradient throughput* (Gbps); (iii) the *CPU usage* (%); (iv) the *one-hop latency* (μs); and (v) the *packet loss rate* (‰). We count the number of gradient packets and record the transmission time to calculate the *packet processing rate* and *gradient throughput*. On the one hand, we measure the impact of different gradient payload sizes in a packet on the metrics (i) and (ii), so as to determine the optimal gradient packet size. On the other hand, we adopt metrics (ii) and (iii) to show the linear scaling performance and on-demand CPU resources occupation of XDP-based aggregation.
   We measure the round-trip delay of worker-server-PS and worker-PS separately, and then calculate their difference to derive the one-hop processing latency, *i.e.*, metric (iv). We count the number of retransmitted gradient packets to obtain the *packet loss rate*, that is, metric (v).
2) To evaluate the aggregation acceleration performance of XAgg in heterogeneous testbed scenarios, we measure (i) the *per-epoch communication time* (s); (ii) the *per-epoch time* (s); (iii) the *peak throughput* (Gbps) of workers and (iv) the *training throughput* (images/s) of workers; and (v) the *test accuracy* (%).
   We measure the *per-epoch communication time* from the worker sending its local gradients to receiving the aggregated gradients in each epoch. We record the time between two consecutive epochs as the *per-epoch time*. Moreover, we use iftop [44] to monitor the egress traffic of workers, and obtain the *peak throughput* of workers. Then, we can calculate the *training throughput* of workers based on metric (i), *i.e.*, the number of images trained per second. Finally, we record the *test accuracy* of XAgg over time.

Benchmarks. We compare XAgg with the other four benchmarks. The first one is similar to XAgg, called TCP-Agg,

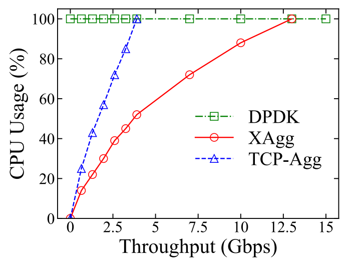


Fig. 9. CPU usage vs. Throughput.

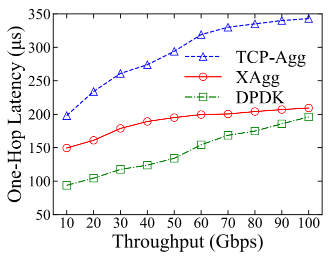


Fig. 10. One-Hop latency vs. Throughput.

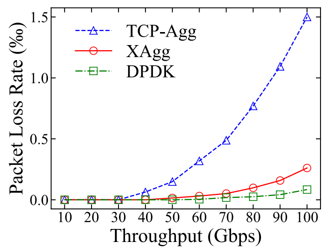


Fig. 11. Packet loss rate vs. Throughput.

which also uses idle servers for gradient pre-aggregation. The only difference is that this scheme adopts native kernel TCP stack for gradient transmission. The second benchmark is an in-network aggregation framework called SwitchML [13]. It leverages the programmable switches to aggregate the model gradients from multiple workers, thereby reducing the volume of exchanged data and eliminating the bandwidth bottleneck of the PS. To control the variables and ensure the comparison fairness, the SwitchML worker adopts the raw socket to send SwitchML format packets, and also utilizes XDP to receive aggregated gradients from P4 switches, which is similar to the implementation of XAgg. The third one is the most popular All-Reduce architecture NVIDIA Collective Communications Library (NCCL) [45]. The last one is BytePS [21], which is a PS-based acceleration framework.

B. Microbenchmark

We evaluate the baseline performance of XAgg, TCP-Agg, and DPDK to demonstrate that XDP has higher performance compared with native TCP and lower CPU usage compared with DPDK. Thus providing a basis for the technology selection of aggregation data plane.

The Optimal Gradient Packet Size. We first measure the impact of different gradient sizes in a packet on the *packet processing rate* of XAgg, DPDK and TCP-Agg with a single CPU core. Specifically, we use three machines for this experiment, including a worker, an server, and a PS. The number of RX queues on the server is set to 1, so that the aggregator will use one CPU core. We send gradient traffic from the worker to the server, where each packet payload contains 32 to 320 gradient values, *i.e.*, 128-1280 bytes. After receiving the gradient packets, the XDP aggregator on the server will perform the aggregation operation and forward the aggregated gradients to the PS. We count the number of received packets on the PS to calculate the packet processing rate of the XDP aggregator.

As shown in Fig. 6, the processing capacity of the aggregator decreases as the gradient payload size increases. For example, XAgg processes 4.19M packets per second with the payload size of 32. When the gradient payload size scales to 320, the packet processing rate reduces to 1.11Mpps. That is because the aggregator needs to traverse the gradient array of each packet. A larger gradient payload size will increase the processing time of a single packet, thereby reducing the packet processing rate. Furthermore, we observe 2.77-3.54× performance improvement of XDP compared to TCP under the same gradient size. It means that XDP can effectively improve the throughput of gradient aggregation and forwarding by pre-stack packet processing. Meanwhile, the single core processing performance of DPDK is slightly higher than that of XDP by 10.3% -23.1%. This is because DPDK adopts busy polling to pull packets from NIC to the user space for processing [30].

Although the aggregator achieves a higher processing rate with a smaller packet size, this increases the processing overhead of the packet header and reduces the goodput. Therefore, XAgg may not achieve the optimal throughput when a packet carries a few gradient values. We need to further measure the gradient throughput under different packet sizes to determine the optimal gradient payload size. By Fig. 7, when a packet carries 256 gradient values, the single-core throughput of XAgg will reach a maximum of 13.03 Gbps. Meanwhile, TCP-Agg's throughput of a single core reaches 4.12 Gbps with 320-element gradient payload. As a result, we select 256-element gradient payload in XAgg and conduct subsequent experiments.

Linear Scaling Performance. Fig. 8 shows that the *throughput* of XAgg will scale linearly with the increasing number of CPU cores. Specifically, given a packet of 256 gradient values, the aggregate bandwidth of XAgg can reach 13 Gbps. When scaling to 10 cores, XAgg will hit the NIC's bandwidth limit of the machine with throughput at 99.5 Gbps. DPDK performs slightly better, so its single core bandwidth is 14.9 Gbps, and it can reach 100 Gbps bandwidth with 9 CPU cores. Meanwhile, although the processing capacity of TCP-Agg also increases with the number of CPU cores, its overall performance lags significantly behind that of XAgg. For example, the throughput of XAgg and TCP-Agg with 10 CPU cores is 99.5 Gbps and 29.8 Gbps respectively, and the throughput of XAgg is 3.3× higher than that of TCP-Agg. From the above results, we can conclude that XDP can enable servers to utilize 10 CPU cores for gradient aggregation at a line rate of 100 Gbps, which significantly improves the in-network aggregation performance of XAgg.

CPU Usage. Fig. 9 illustrates the CPU usage with bandwidth growth for the three schemes. We use the mpstat tool to measure the CPU usage when performing gradient aggregation on a single core. Since DPDK adopts busy polling to process the packets, its CPU usage is always pegged at 100%. In contrast, both XDP and TCP adopt soft interrupt processing, which can smoothly scale CPU usage with the increasing load [29]. Although the performance of XDP is slightly weaker than that of DPDK, its on-demand CPU resources occupation is obviously more suitable for deployment on idle servers.

One-Hop Latency. We then analyze the impact of deploying XDP-based aggregators on *one-hop latency* on servers. Based on the previous results, we allocate 10 CPU cores to the aggregator. To obtain the processing latency of one hop in XAgg, we measure the round-trip delay of worker-server-PS and worker-PS separately, and then calculate their difference. In Fig. 10, the average one-hop latency increases with throughput, as a larger volume of traffic will bring higher queuing delay at the switches and NICs. For example, the one-hop latency of XAgg at 10 Gbps throughput is 149.5μs, and this latency increases by 40% to 209.3μs at 100 Gbps. Meanwhile, we observe that XAgg reduces the one-hop latency by 24%-39% compared with TCP-Agg, which means that XAgg can effectively accelerate gradient aggregation based on TCP protocol stack. The one-hop latency of DPDK is 6% -37% lower than that of XAgg, because the busy polling of DPDK outperforms the soft interrupt of XDP in terms of latency.

Packet Loss Rate. We further calculate the *packet loss rate* of three schemes under different throughputs, which aims to prove that XDP has better reliability and aggregation performance than TCP-based aggregation. For XAgg and DPDK, we do not enable its reliable transmission function. In addition, considering the reliable transmission of TCP protocol, we take the timeout retransmission ratio of the worker as the packet loss rate of TCP-Agg. As shown in Fig. 11, the packet loss rate of XAgg is always lower than that of TCP-Agg. For example, the packet loss rate of TCP at 100Gbps is 1.5‰, while the packet loss rate of XDP is only 0.261‰, which is 87% lower than that of TCP. Moreover, the packet loss rate will increase slightly only when the gradient throughput approaches or reaches the bandwidth limit of NIC. Packet loss comes from the overflow of the NIC RX buffer. When massive packets arrive, the program cannot timely process the received packets, resulting in Rx buffer overflow and packet loss.

We conclude from the above experimental results that XAgg has higher performance compared with TCP-Agg and lower resource usage compared with DPDK. Thus, we choose XDP as the data plane technology for gradient aggregation.

C. Testbed Evaluation

Testbed Setups. We use 10 servers to build the XAgg testbed, including 7 workers, 2 idle servers and 1 PS. Specifically, each server is equipped with 44 cores of Intel Xeon 6152 processor, 128 GB memery. All the servers run Ubuntu 18.04 with Linux kernel 5.4. Each worker runs PyTorch [37] for local model training. The topology of the XAgg testbed is shown in Fig. 12, where the gradients of $W_1 - W_3$ and $W_4 - W_6$ are aggregated on servers S_1 and S_2, respectively, while the gradients of S_1, S_2, and W_7 are aggregated on PS. In addition, the computation and bandwidth of the 7 workers are heterogeneous. $W_1 - W_3$ are equipped with NVIDIA RTX 3080 GPU and Intel XL710 40G NIC, while $W_4 - W_7$ are equipped with NVIDIA RTX 3090 GPU and Mellanox ConnectX-6 100G NIC.

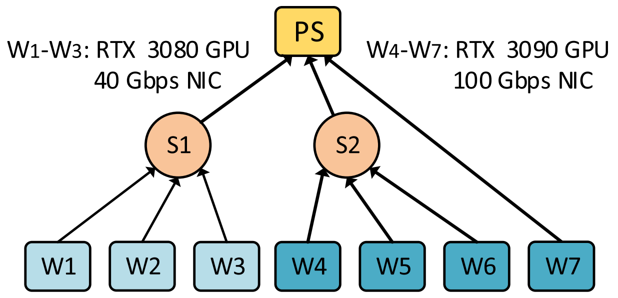


Fig. 12. The testbed is configured with 7 workers, 2 idle servers and 1 PS. The gradients of $W_1 - W_3$ and $W_4 - W_6$ are aggregated on servers S_1 and S_2, respectively. The gradients of S_1, S_2 and W_7 are aggregated on PS. Moreover, $W_1 - W_3$ are equipped with RTX 3080 GPU and 40 Gbps NIC, while $W_4 - W_7$ are equipped with RTX 3090 GPU and 100 Gbps NIC.

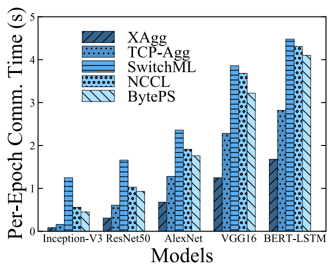


Fig. 13. Per-Epoch communication time under different models.

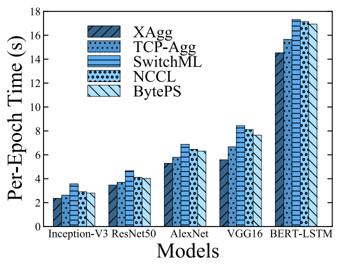


Fig. 14. Per-epoch time under different models.

When testing the comparison benchmarks, TCP-Agg and XAgg share the same topology. In the BytePS framework, 7 workers are directly connected to PS. The 7 workers of SwitchML are directly connected to a Wedge100BF-32x programmable switch with an Intel Tofino chip [23], which is then connected to PS.

To evaluate the gradient aggregation acceleration performance of XAgg in heterogeneous scenarios, we choose four CNN models and one Transformer model with different sizes. The CNN models include Inception-V3 [46], ResNet50 [1], AlexNet [47] and VGG16 [48]. We train these models on Cifar-10 dataset [49], which contains 60000 images, 50000 of them for training and 10000 of them for testing, labeled in 10 classes. The Transformer model is the BERT-BiLSTM-CRF model [50], [51] for named entity recognition (NER)

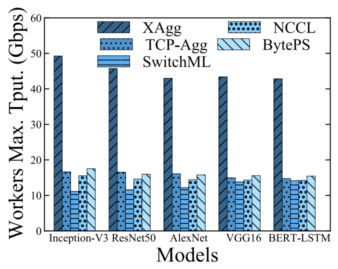


Fig. 15. Peak throughput of workers under different models.

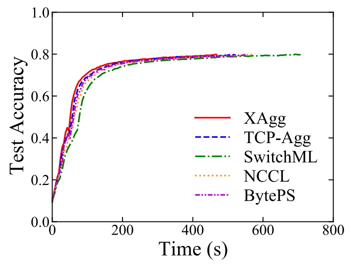


Fig. 16. Test accuracy over time under inception-V3.

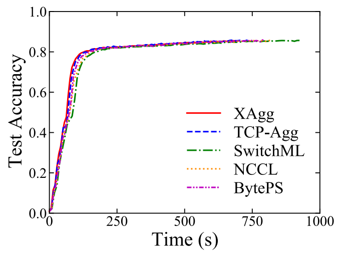


Fig. 17. Test accuracy over time under ResNet50.

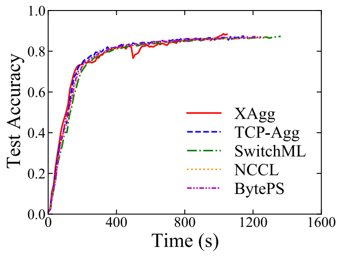


Fig. 18. Test accuracy over time under AlexNet.

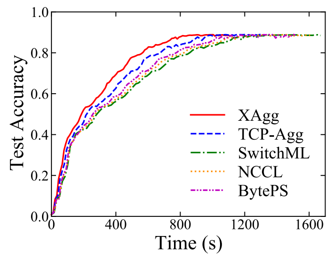


Fig. 19. Test accuracy over time under VGG16.

task [52]. We adopt CLUENER-2020 dataset [53] to train this model, which contains 12091 sentences, 10748 of them for training and 1343 of them for testing. The sizes of the above five models are 25MB, 97MB, 233MB, 528MB and 542MB respectively. XAgg can completely store the above models on host, but the limited on-chip memory of programmable

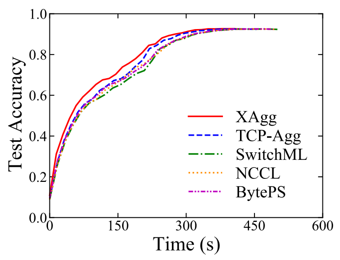


Fig. 20. Test accuracy over time under BERT-LSTM.

switches will seriously affect the aggregation of large models. Moreover, we set the training epoch number of CNN models to 200 and the Transformer model to 30.

Per-Epoch Communication Time. In this set of evaluations, we focus on the acceleration effect of XAgg on per-epoch training (Figs. 13-14). We define the *communication time* as the period from the beginning of workers sending the gradient to receiving the aggregated gradient. Fig. 13 shows the per-epoch communication time of different schemes when training the four DNN models. For ResNet50, the communication time of XAgg, TCP-Agg, SwitchML, NCCL and BytePS is 0.31s, 0.61s, 1.66s, 1.03s and 0.93s, respectively. The per-epoch communication time of XAgg is reduced by 49%, 82%, 70% and 66% compared to that of TCP-Agg, SwitchML, NCCL and BytePS, respectively. The reason is that XAgg uses XDP to achieve efficient gradient aggregation, avoid the protocol stack overhead, and eliminate the communication bottleneck at the PS. In contrast, SwitchML will block for a period of time due to the heterogeneous computation and bandwidth of workers before reaching the peak aggregation performance.

Per-Epoch Time. We further evaluate the impact of the reduction in communication time on the total training time of one epoch. We record the *per-epoch time*, which includes the local training time and communication time of workers in one epoch. Although our scheme does not optimize the local training time, the acceleration of gradient aggregation can effectively speed up of each epoch of training. We observe from Fig. 14 that the per-epoch training time of XAgg is always the lowest compared with that of other four benchmarks. In addition, XAgg has a more significant acceleration effect on communication-intensive large-size models. For example, compared to SwitchML, XAgg can reduce the per-epoch time by 23%, 26% and 34% under AlexNet, ResNet50 and VGG16 models, respectively. Because the BERT-LSTM model has a relatively low proportion of communication time (25%), the effect of XAgg in optimizing its communication time is not significant. However, the per-epoch training time of XAgg is still lower compared with that of other four benchmarks. For example, XAgg can still reduce the per-epoch time of BERT-LSTM model by 16% compared to SwitchML.

Peak Gradient Throughput of Workers. We use iftop [44] to monitor the egress traffic of workers and record the *peak gradient throughput*. As shown in Fig. 15, XAgg always achieves the highest gradient throughput compared to other schemes when training these five models. For example, when training ResNet50, the workers' maximum throughput of XAgg, TCP-Agg, SwitchML, NCCL and BytePS is 45.7 Gbps, 16.6 Gbps, 11.6 Gbps, 14.6 Gbps and 16 Gbps. It means that XAgg can increase the gradient throughput by 2.8×, 3.9×, 3.1× and 2.9× compared with the above schemes, respectively. The reason is that XAgg uses idle resources on servers for gradient pre-aggregation, and the abundant memory prevents the gradient sending of workers from blocking in heterogeneous scenarios. In addition, the rate negotiation mechanism of XAgg can effectively improve the bandwidth usage of workers. On the contrary, the peak gradient throughput of workers in SwitchML is only 11.6 Gbps, which is obviously far lower than expected. This is because the workers in SwitchML will not send subsequent gradients until they receive the previous aggregated results, which severely degrades the gradient sending rate.

Test Accuracy over Time. In this set of evaluations, we evaluate the *test accuracy* of XAgg over time under these five DNN models, as shown in Figs. 16-20. The experimental results show that XAgg takes the least time to complete training and achieves a similar test accuracy compared with other alternatives. We observe from Fig. 19 that when training VGG16, XAgg first completes 200 epochs training in 1116s with an accuracy of 0.884, while the total training time for TCP-Agg, SwitchML, NCCL and BytePS is 1334s, 1688s, 1622s and 1530s, respectively. From the above experimental data, we can calculate that XAgg can speed up the distributed model training by 1.17×, 1.34×, 1.31× and 1.27× compared with TCP-Agg, SwitchML, NCCL and BytePS. In addition, XAgg completes 30 epochs training of BERT model training within 435s and achieved an accuracy of 0.924, which reduced the training time by 8%, 16%, 15% and 14% compared to other four benchmarks, respectively. As a result, we conclude that by utilizing XDP-based aggregators, XAgg accelerates gradient aggregation and eliminates the communication bottleneck in heterogeneous scenarios without affecting the training accuracy.

Evaluation of Delays. onsidering that the aggregation tree topology may occasionally change during training, we evaluate the impact of the aggregator deployment delay on training efficiency. XDP-based aggregators depend on specific kernel versions. Generally, servers in the data center are equipped with consistent kernel versions, so XAgg's aggregator can be compiled once and run everywhere (CO-RE) [54]. In cases of different kernel versions, aggregators need to be compiled on each node before deployment. Deployment latency with and without precompilation is 0.34s and 0.68s, respectively. We can conclude that aggregator deployment will not be a bottleneck in XAgg.

Moreover, the rate initialization delay of worker, server and PS is 12ms, 5ms and 2ms, respectively. The rate update delay of worker and server is 2ms and 1ms respectively. We can conclude that the delay of rate negotiation is negligible compared to the training/gradient transmission time.

We then record the real-time transmission rate of the worker based on XAgg and TCP-Agg when the bandwidth decreases. Specifically, we set the initial transmission rate to 60 Gbps and then reduce the receiving bandwidth of the aggregation node

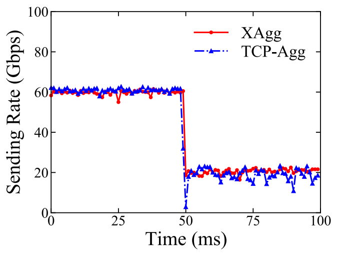


Fig. 21. Sending rate over time under bandwidth decrease.

to 20 Gbps. We record the sending rate of workers every 10ms, as shown in Fig. 21. XAgg adopts the group-based acknowledgment mechanism, which can achieve the same bandwidth adjustment effect as TCP, but the number of acknowledgement packets is much smaller than that of TCP. In addition, when the bandwidth decreases, TCP will reset cwnd to 1 and re-execute the Slow-Start algorithm, and its bandwidth will sharply decrease to 0. The bandwidth adjustment of XAgg is smoother than that of TCP.

VII. Related Work

There are two typical paradigms for scaling out distributed model training: model parallelism and data parallelism [55]. Model parallelism refers to dividing a large model into multiple layers and allocating these layers to several workers. Each worker calculates the gradient of the assigned layer(s) [56]. In this paper, we focus on accelerating data parallelism model training, which splits the whole dataset into multiple workers [57]. Specifically, in each iteration, the workers train the model with their own dataset partitions and generate local gradients. Subsequently, workers communicate with each other and exchange local model gradients to obtain global gradients, also called gradient aggregation.

Typical Gradient Aggregation Framework. Parameter Server (PS) [5] and AllReduce (AR) [58] are two widely used gradient aggregation frameworks for data parallelism training. PS usually contains two kinds of nodes: workers and parameter servers (PSs). Specifically, in each iteration, the workers first train the model and transmit their local gradients to the PS(s) for aggregation. After that, the PS(s) will send the aggregated gradient results back to each worker. Obviously, as the model size and the number of workers increase, the bandwidth of the centralized PS(s) will become the performance bottleneck of distributed model training. AR is proposed to alleviate the communication bottleneck of the PS, which adopts collective communication for gradient aggregation. We take the widely used Ring-AR as an example, where all nodes are workers, and organized in a ring topology [59]. Each worker sends a gradient partition to its successor and receives another partition from its predecessor. Although AR can achieve optimal bandwidth utilization in homogeneous scenarios, but it suffers from poor robustness, especially in heterogeneous scenarios. The speed decrease or failure of a node in the ring will reduce the throughput of the whole ring. Optimizing Ring AllReduce in heterogeneous scenarios requires designing new communication mechanisms, such as constructing multiple rings [60], [61].

Communication Optimization. Prior works attempt to speed up the communication process for distributed model training through two approaches: gradient compression and communication scheduling. Many studies [12], [14], [15], [16] have proposed compressing gradients to reduce traffic while guaranteeing training convergence. There are two main compression techniques: sparsification [12], [14] and quantization [15]. For example, OmniReduce [12] leverages gradient sparsity and only sends non-zero data blocks to minimize traffic transfer. However, gradient compression schemes suffer from a decrease of training accuracy [13], [62]. Besides reducing gradient transfer, many works try to optimize communication scheduling, including designing high-performance traffic schedulers [17], [18], [19], and building training and communication pipelines [20], [21]. For instance, Geryon [19] determines the scheduling scheme for gradient flows according to their priorities to maximize the utilization of bandwidth resources. BytePS [21] transmits gradients in layers, increasing the overlap between workers training and network transmission through fine-grained communication. However, these efforts cannot reduce the traffic volume while performing gradient aggregation, and may still encounter the communication bottleneck on links.

In-network Aggregation. The idea of in-network aggregation was first explored in the field of wireless networks [63] and big data [64]. For example, work [63] aggregates information between the paths from sensor nodes towards base stations to reduce data transmission in wireless sensor networks and prolong network lifetime. NetAgg [64] uses switch-attached middleboxes to aggregate data from MapReduce and search engines. As communication becomes a bottleneck in distributed model training, many efforts [13], [22], [65] have attempted to implement in-network gradient aggregation to accelerate distributed training with programmable hardwares. SwitchML [13] and ATP [22] offload gradient aggregation logic to Tofino-based programmable switches to alleviate the communication bottlenecks of the PS. However, the limited on-chip memory of programmable switches is difficult to adapt to the heterogeneous distributed training system, resulting in low aggregation performance and even degradation to the PS framework. iSwitch [65] leverages FPGA-based programmable switches to aggregate gradients for reinforcement learning. Although FPGAs have larger on-chip memory, it operates at much lower bandwidth (4×10 Gbps).

On-host Network Acceleration Technology. Although the programmable switch provides high bandwidth and high-performance processing/forwarding, its limited on-chip memory cannot store the complete model gradients. Therefore, the programmable switch requires high synchronization of gradient transmission, making it difficult to adapt to heterogeneous distributed training systems. In contrast, on-host software network acceleration technologies provide sufficient memory and flexible development and deployment capabilities, including DPDK [35], XDP [29], [30], Netmap [66], etc. DPDK adopts the kernel bypass approach to realize a high-performance and low-latency packet processing plane in user space [35]. However, DPDK is not suitable to deploy on servers for in-network gradient aggregation. First, DPDK completely takes over the network device from the kernel, which leads to potential security risks, as well as affects the processing of regular traffic [30]. Second, the busy polling mechanism of DPDK needs to occupy dedicated CPU core(s) [30], which will bring unnecessary waste of resources to servers. To this end, we seek a high-performance and low-overhead on-host network acceleration technology to adapt to the gradient aggregation with idle resources on servers.

eBPF/XDP and Related Applications. Extended Berkeley Packet Filter (eBPF) is a highly flexible and efficient virtual machine-like construct inside the Linux kernel [67]. Developers can inject custom code into the kernel through various hook points in a safe manner. eXpress Data Path (XDP) is one of the most important eBPF hooks for high-performance data path that can perform packet processing before the kernel network stack [29], [30]. XAgg utilizes XDP to implement gradient aggregation and forwarding on servers because of its linear scaling performance, low processing latency, on-demand requests for CPU resources, and allowing regular traffic to be processed normally by the network stack. In addition to networking, eBPF is currently widely used for security [68], monitoring [69] and storage [70], [71]. For example, BMC [70] constructs a in-memory key-value store accelerator, which adopts XDP for pre-stack requests processing and uses eBPF maps as in-kernel cache.

VIII. Discussion

XAgg Applicable Scenarios. Considering some dedicated GPU clusters may not have many idle CPU and memory resources. For example, Microsoft uses dedicated GPU cluster with full CPU/memory saturation to train large-scale GPT models. XAgg is more suitable for public cloud scenarios. Specifically, cloud providers can implement XAgg as an acceleration service for tenants who cannot afford large-scale dedicated GPU clusters. Tenants can rent a small number of PS nodes and use XAgg's accelerated services on demand to speed up training while saving costs.

Synchronous and Asynchronous Update. Synchronous gradient updates can solve the problem of straggler workers in heterogeneous scenarios to some extent [72], [73]. However, asynchronous gradient update suffers from the stale gradients problem that degrade convergence, resulting in an almost no improvement in time to convergence [74]. In this paper, considering that P4-based in-network aggregation cannot adapt to heterogeneous scenarios with synchronous aggregation, XAgg is proposed to optimize synchronous aggregation under computation and bandwidth heterogeneity.

Hyperparameter optimization. Hyperparametric optimization, such as adaptive batch size, can enable workers to start sending gradients almost simultaneously, which is beneficial for heterogeneous computation scenarios. However, this method is difficult to deal with bandwidth heterogeneity effectively. For example, in the INA scenario, the aggregation rate of the programmable switch is still limited to the slowest bandwidth, even if all workers start sending gradients simultaneously with adaptive batch size. Our paper comprehensively consider the computation and bandwidth heterogeneity. Considering that our work can optimize the bandwidth heterogeneity and adaptive batch size can optimize the computational power heterogeneous scenario, XAgg can work together with adaptive batch size.

IX. Conclusion

This paper focuses on achieving high-performance and stable gradient aggregation under heterogeneous distributed model training. We propose XAgg, which deploys the XDP-based aggregator on servers in data centers to accelerate heterogeneous gradient aggregation. Specifically, the abundant idle memory on servers can cache the entire gradient and wait for stragglers to catch up, so as to deal with computation/bandwidth heterogeneity. Moreover, XDP can provide high-performance and low-latency gradient pre-aggregation. We conduct microbenchmark and testbed with real-world DNN models and datasets. Experimental results show that XAgg can achieve gradient aggregation and forwarding at 100 Gbps with 10 CPU cores, and its performance is 3.3× higher than that of TCP-based aggregation. In addition, XAgg reduces communication time by 49%-82% compared with state-of-the-art solutions.

References

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

[2] A. Vaswani et al., "Attention is all you need," in *Proc. Adv. neural Inf. Process. Syst.*, vol. 30, 2017, pp. 1–12.

[3] S. Zhang, L. Yao, A. Sun, and Y. Tay, "Deep learning based recommender system: A survey and new perspectives," *ACM Comput. Surv.*, vol. 52, no. 1, pp. 1–38, Jan. 2020.

[4] C. Sun, A. Shrivastava, S. Singh, and A. Gupta, "Revisiting unreasonable effectiveness of data in deep learning era," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Oct. 2017, pp. 843–852.

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

[6] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis, "In-network computation is a dumb idea whose time has come," in *Proc. 16th ACM Workshop Hot Topics Netw.*, Nov. 2017, pp. 150–156.

[7] NVIDIA. (2020). *NVIDIA Ampere Architecture in-Depth*. [Online]. Available: https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

[8] Google. (2022). *Cloud TPU: Train and Run Machine Learning Models Faster Than Ever Before*. [Online]. Available: https://cloud.google.com/tpu

[9] Habana Gaudi. (2022). *High Performance Training and Inference*. [Online]. Available: https://habana.ai

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

[11] NVIDIA. (2022). *NVIDIA Data Center Deep Learning Product Performance*. [Online]. Available: https://developer.nvidia.com/deep-learning-performance-training-inference

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

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

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

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

[16] W. Wen et al., "TernGrad: Ternary gradients to reduce communication in distributed deep learning," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, pp. 1508–1518.

[17] S. H. Hashemi, S. A. Jyothi, and R. Campbell, "TicTac: Accelerating distributed deep learning with communication scheduling," in *Proc. Mach. Learn. Syst.*, vol. 1, 2019, pp. 418–430.

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

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

[20] Y. Huang et al., "GPipe: Efficient training of giant neural networks using pipeline parallelism," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 103–112.

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

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

[23] Intel. (2022). *Intel Tofino*. [Online]. Available: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html

[24] C. Jiang, G. Han, J. Lin, G. Jia, W. Shi, and J. Wan, "Characteristics of co-allocated online services and batch jobs in internet data centers: A case study from Alibaba cloud," *IEEE Access*, vol. 7, pp. 22495–22508, 2019.

[25] J. Guo et al., "Who limits the resource efficiency of my datacenter: An analysis of Alibaba datacenter traces," in *Proc. IEEE/ACM 27th Int. Symp. Quality Service (IWQoS)*, Jun. 2019, pp. 1–10.

[26] Q. Cai, S. Chaudhary, M. Vuppalapati, J. Hwang, and R. Agarwal, "Understanding host network stack overheads," in *Proc. ACM SIGCOMM Conf.*, Aug. 2021, pp. 65–77.

[27] K. Yasukata, M. Honda, D. Santry, and L. Eggert, "StackMap: Low-latency networking with the OS stack and dedicated NICs," in *Proc. USENIX Annu. Tech. Conf.*, 2016, pp. 43–56.

[28] Y. Zhang et al., "ParaBox: Exploiting parallelism for virtual network functions in service chaining," in *Proc. Symp. SDN Res.*, Apr. 2017, pp. 143–149.

[29] T. Høiland-Jørgensen et al., "The express data path: Fast programmable packet processing in the operating system kernel," in *Proc. 14th Int. Conf. Emerg. Netw. Exp. Technol.*, Dec. 2018, pp. 54–66.

[30] M. A. M. Vieira, M. S. Castanho, R. D. G. Pacífico, E. R. S. Santos, E. P. M. C. Júnior, and L. F. M. Vieira, "Fast packet processing with eBPF and XDP: Concepts, code, challenges, and applications," *ACM Comput. Surv.*, vol. 53, no. 1, pp. 1–36, Jan. 2021.

[31] S. Banerjee, B. Bhattacharjee, and C. Kommareddy, "Scalable application layer multicast," in *Proc. 2002 Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, 2002, pp. 205–217.

[32] J. Robinson, M. Singh, R. Swaminathan, and E. Knightly, "Deploying mesh nodes under non-uniform propagation," in *Proc. IEEE INFOCOM*, Mar. 2010, pp. 1–9.

[33] H. Pan et al., "Dissecting the communication latency in distributed deep sparse learning," in *Proc. ACM Internet Meas. Conf.*, Oct. 2020, pp. 528–534.

[34] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle, "A study of network stack latency for game servers," in *Proc. 13th Annu. Workshop Netw. Syst. Support Games*, Dec. 2014, pp. 1–6.

[35] DPDK. (2022). *Data Plane Development Kit*. [Online]. Available: https://www.dpdk.org/

[36] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle, "Comparison of frameworks for high-performance packet IO," in *Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, May 2015, pp. 29–38.

[37] A. Paszke et al., "Pytorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 8026–8037.

[38] S. Luo, H. Xing, and K. Li, "Near-optimal multicast tree construction in leaf-spine data center networks," *IEEE Syst. J.*, vol. 14, no. 2, pp. 2581–2584, Jun. 2020.

[39] L. C. Lau and M. Singh, "Additive approximation for bounded degree survivable network design," in *Proc. 40th Annu. ACM Symp. Theory Comput.*, May 2008, pp. 759–768.

[40] J. Byrka, F. Grandoni, T. Rothvoß, and L. Sanitß, "An improved LP-based approximation for Steiner tree," in *Proc. 42nd ACM Symp. Theory Comput.*, Jun. 2010, pp. 583–592.

[41] M. Alizadeh and T. Edsall, "On the data path performance of leaf-spine datacenter fabrics," in *Proc. IEEE 21st Annu. Symp. High-Perform. Interconnects*, Aug. 2013, pp. 71–74.

[42] M. Dalton et al., "Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization," in *Proc. 15th USENIX Symp. Networked Syst. design Implement.*, 2018, pp. 373–387.

[43] Z. Wang, H. Xu, J. Liu, H. Huang, C. Qiao, and Y. Zhao, "Resource-efficient federated learning with hierarchical aggregation in edge computing," in *Proc. IEEE Conf. Comput. Commun.*, May 2021, pp. 1–10.

[44] Iftop. (2022). *Iftop: Display Bandwidth Usage on an Interface*. [Online]. Available: http://www.ex-parrot.com/~pdw/iftop/

[45] NVIDIA NCCL. (2023). *NVIDIA Collective Communications Library (NCCL)*. [Online]. Available: https://developer.nvidia.com/nccl

[46] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 2818–2826.

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

[48] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.

[49] A. Krizhevsky, V. Nair, and G. Hinton. (2022). *The Cifar-10 Dataset*. [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html

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

[51] 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 *Proc. 12th Int. Congr. Image Signal Process., Biomed. Eng. Informat. (CISP-BMEI)*, Oct. 2019, pp. 1–5.

[52] J. Li, A. Sun, J. Han, and C. Li, "A survey on deep learning for named entity recognition," *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 1, pp. 50–70, Jan. 2022.

[53] L. Xu et al., "CLUENER2020: Fine-grained named entity recognition dataset and benchmark for Chinese," 2020, *arXiv:2001.04351*.

[54] A. Nakryiko. (2020). *BPF Portability and CO-RE*. [Online]. Available: https://facebookmicrosites.github.io/bpf/blog/2020/02/19/bpf-portability-and-co-re.html

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

[56] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-LM: Training multi-billion parameter language models using model parallelism," 2019, *arXiv:1909.08053*.

[57] C. J. Shallue, J. Lee, J. Antognini, J. Sohl-Dickstein, R. Frostig, and G. E. Dahl, "Measuring the effects of data parallelism on neural network training," 2018, *arXiv:1811.03600*.

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

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

[60] X. Jia et al., "Highly scalable deep learning training system with mixed-precision: Training ImageNet in four minutes," 2018, *arXiv:1807.11205*.

[61] J. Geng, D. Li, Y. Cheng, S. Wang, and J. Li, "HiPS: Hierarchical parameter synchronization in large-scale distributed machine learning," in *Proc. Workshop Netw. Meets AI ML* , 2018, pp. 1–7.

[62] S. Wang, A. Pi, X. Zhou, J. Wang, and C.-Z. Xu, "Overlapping communication with computation in parameter server for scalable DL training," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 9, pp. 2144–2159, Sep. 2021.

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

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

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

[66] L. Rizzo, "netmap: A novel framework for fast packet I/O," in *Proc. 21st USENIX Secur. Symp.*, 2012, pp. 101–112.

[67] Cilium. (2022). *BPF and XDP Reference Guide*. [Online]. Available: https://docs.cilium.io/en/stable/bpf/

[68] Falco. (2022). *Cloud Native Runtime Security*. [Online]. Available: https://falco.org/

[69] Netdata. (2022). *Netdata: Monitoring and Troubleshooting Transformed*. [Online]. Available: https://www.netdata.cloud/

[70] Y. Ghigoff, J. Sopena, K. Lazri, A. Blin, and G. Müller, "BMC: Accelerating memcached using safe in-kernel caching and pre-stack processing," in *Proc. 18th USENIX Symp. Networked Syst. Design Implement.*, 2021, pp. 487–501.

[71] Y. Zhong et al., "XPR: In-kernel storage functions with eBPF," in *Proc. 16th USENIX Symp. Operating Syst. Design Implement.*, 2022, pp. 375–393.

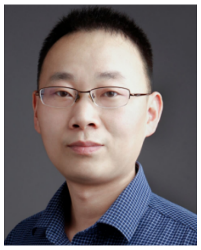
[72] T. Chen et al., "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," 2015, *arXiv:1512.01274*.

[73] J. Dean et al., "Large scale distributed deep networks," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 25, 2012, pp. 1223–1231.

[74] M. Abadi et al., "TensorFlow: A system for large-scale machine learning," in *Proc. OSDI*, vol. 16. Savannah, GA, USA, 2016, pp. 265–283.

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.

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.



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

Peng Yang is currently pursuing the Ph.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 distributed training.