Canary: Decentralized Distributed Deep Learning Via Gradient Sketch and Partition in Multi-Interface Networks

Qihua Zhou, *Student Member, IEEE*, Kun Wang, *Senior Member, IEEE*, Haodong Lu, *Student Member, IEEE*, Wenyao Xu, *Senior Member, IEEE*, Yanfei Sun, *Member, IEEE*, and Song Guo, *Fellow, IEEE*

Abstract—The multi-interface networks are efficient infrastructures to deploy distributed Deep Learning (DL) tasks as the model gradients generated by each worker can be exchanged to others via different links in parallel. Although this decentralized parameter synchronization mechanism can reduce the time of gradient exchange, building a high-performance distributed DL architecture still requires the balance of communication efficiency and computational utilization, i.e., addressing the issues of traffic burst, data consistency, and programming convenience. To achieve this goal, we intend to asynchronously exchange gradient pieces without the central control in multi-interface networks. We propose the *Piece-level Gradient Exchange* and *Multi-interface Collective Communication* to handle parameter synchronization and traffic transmission, respectively. Specifically, we design the *gradient sketch* approach based on 8-bit uniform quantization to compress gradient tensors and introduce the *colayer* abstraction to better handle gradient partition, exchange and pipelining. Also, we provide general programming interfaces to capture the synchronization semantics and build the *Gradient Exchange Index* (GEI) data structures to make our approach online applicable. We implement our algorithms into a prototype system called Canary by using PyTorch-1.4.0. Experiments conducted in Alibaba Cloud demonstrate that Canary reduces 56.28 percent traffic on average and completes the training by up to 1.61x, 2.28x, and 2.84x faster than BML, Ako on PyTorch, and PS on TensorFlow, respectively.

Index Terms—Distributed systems, multi-interface network, deep learning, gradient sketch, decentralized architecture

1 Introduction

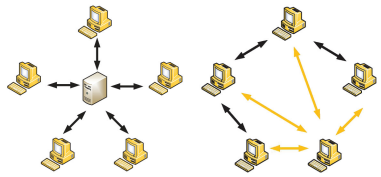
The centralized parameter server (PS) architecture [1], [2], [3] has achieved great success to operate large-scale distributed Deep Learning (DL) tasks [4], [5], [6], [7], [8], [9] and is widely used in system implementation (e.g., Poseidon [2], Petuum [10] and MXNet [11]). However, the *server-worker* structure in PS architecture is often subject to the communication pattern of frequent aggregation and broadcast, where the logical central servers may be the bottleneck and suffer from severe traffic pressure with the increment of cluster scale [12]. Moreover, commodity PS-based frameworks are often deployed in FatTree [13] network, which cannot well match the communication requirements of parallel DL [14], especially when the available bandwidth is limited and computational heterogeneity exists.

Observing the limitations of PS-based architectures, a growing body of researches [12], [14], [15], [16] have studied how to handle distributed DL training in a decentralized manner, where machines exchange parameters to others directly without the central control. The recently presented BML [14] architecture working in the BCube network [17] is a pertinent case to practice the decentralized DL training, with less global synchronization time over commodity PS architectures. The success of BML comes from the traffic transmission through different links in parallel, i.e., utilizing the communication capacity of multiple NICs.

However, our preliminary experiments (see Section 3) reveal that existing decentralized approaches have not fully exploited the advantages of multi-interface network, mainly in four aspects: (1) they often follow the scheme developed from *Bulk Synchronous Parallel* (BSP) [2], [14], [18] or the delay-bounded *Stale Synchronous Parallel* (SSP) [12], [19], [20], [21], where the barrier at the end of each synchronization causes the time waste on waiting for the slowest worker [22] and the training performance may suffer from *stragglers* [1], [23]; (2) they have not fully utilized the inherent parallel communication links of multi-interface network because the flow of gradient exchange between two machines is transmitted

- *Qihua Zhou and Yanfei Sun are with the School of Automation and School of Artificial Intelligence, Nanjing University of Posts and Telecommunications12577, Nanjing 210049, China. E-mail: kimizqh@foxmail.com, sunyanfei@njupt.edu.cn.*
- *Kun Wang is with the Department of Electrical and Computer Engineering, University of California, Los Angeles, Los Angeles, CA 90095 USA. E-mail: wangk@ucla.edu.*
- *Haodong Lu is with the School of Internet of Things, Nanjing University of Posts and Telecommunications, Nanjing 210049, China. E-mail: ihaodonglu@gmail.com.*
- *Wenyao Xu is with the Department of Computer Science and Engineering, University at Buffalo, the State University of New York, Buffalo, NY 14260 USA. E-mail: wenyaoxu@buffalo.edu.*
- *Song Guo is with the Department of Computing, Hong Kong Polytechnic University, Hong Kong. E-mail: song.guo@polyu.edu.hk.*

Manuscript received 3 Dec. 2019; revised 1 Aug. 2020; accepted 28 Oct. 2020. Date of publication 9 Nov. 2020; date of current version 24 Nov. 2020. (Corresponding authors: Kun Wang, Yanfei Sun, and Song Guo.) Recommended for acceptance by K. W Cameron. Digital Object Identifier no. 10.1109/TPDS.2020.3036738



(a) Centralized co-ordination. (b) Decentralized coordination.

Fig. 1. Communication in data-parallel DL can be handled via two coordination manners: centralized coordination and decentralized coordination.

TABLE 1
Average Cost Comparison of Layer Characteristics on CPU- and GPU-Equipped Workers With Same DL Training Configuration in Fig. 8

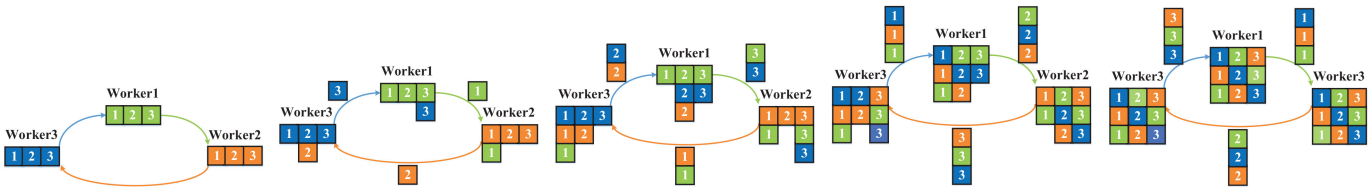
Layer	Float-point Operations	Parameter Dimension	Comp.cpu (ms)	Comp.gpu (ms)	Comm (ms)
CONV	10^8-10^9	10^6	10^4-10^5	1-10^1	10^{-1}-10^1
FC	10^6-10^8	10^8	10^{-1}-10^3	10^{-1}-1	10^2-10^3

via one link at a time (see Fig. 10); (3) they ignore the property of parameter dimension (see Fig. 8 and Table 1) and the distribution of gradient values (see Fig. 9) in different layers, where gradient tensors can be further compressed to reduce traffic volume and communication cost; (4) they are designed for some specific network topologies (e.g., BML is based on BCube network) with different programming methods (e.g., *push/pull* in Fig. 6 and *scatter/gather* in Fig. 10) to handle the distributed communication during DL training. Therefore, transferring a task programmed for a specific network to others (e.g., Tree-based [2], [3], [11] and Mesh-based [12], [24] networks) needs a major reconstruction of codes, which is not convenient for developers [25]. Motivated by these observations, we intend to *design a decentralized DL architecture that fully exploits the advantages of multi-interface networks to accelerate DL training and provides uniform programming interfaces to help developers easily create DL tasks with less code modification*.

This target requires us to conduct an algorithm- and network-level co-design that fully utilizes computational capacity and restricts communication overhead. We achieve it by (1) conducting decentralized gradient exchange in an asynchronous manner, where gradient tensors are compressed and partitioned into pieces for transmitting via different links in parallel, and (2) abstracting the semantics of distributed gradient exchange to improve collective communication methods. In the upper parameter synchronization, we design the *Piece-level Gradient Exchange* (PGE) algorithm to handle DL training via decentralized asynchronous data-parallelism. We propose a *gradient sketch* approach to reduce the volume of gradient tensors with slight precision loss, where the gradients are partitioned into pieces and the corresponding element keys are quantized into *8-bit fixed-point* (INT8) data format, according to the gradient distribution and tensor scale. This approximation approach eliminates frequent traffic burst and reduces the communication time. Moreover, we capture the per-layer property and propose a new abstraction called *colayer*, which is defined as a collection of layers with analogous computation- and communication-level features. By introducing colayer, we can better exploit the power of parallelism and pipelining to overlap communication and computation. In the underlying network communication, we design the *Multi-interface Collective Communication* (MCC) mechanism to fully exploit the transmission parallelism of multi-interface network and exchange gradient pieces asynchronously with staleness bounding. Specifically, observing the low latency and RDMA compatibility in state-of-the-art BML [14], [15], we select the BCube [17] network as a pertinent candidate of multi-interface network and display how to deploy our synchronization mechanism in practice. This mechanism can also apply to other types of multi-interface networks, including Tree-based and Mesh-based topologies.

We implement our training algorithm into Canary a decentralized prototype system that operates parallel DL in multi-interface network and quantizes the gradient pieces into INT8 data format, using a staleness-bounded asynchronous manner. To make our system online applicable, we elaborate two *Gradient Exchange Index* (GEI) data structures to trace all the gradient pieces and globally handle the gradient exchange process. Besides, two hyper-parameters are introduced to restrict the staleness bounding of iteration progress and colayer proportion, so as to guarantee the training convergence. We build Canary in PyTorch-1.4.0 and extract uniform APIs that developers can easily deploy decentralized DL applications in multi-interface networks. We evaluate the performance of Canary in a 34-node cluster by using Alibaba Cloud [26]. To measure the ability of training generalization, we use the training of image classification applications on six models (AlexNet [27], VGG19 [28], Inception-V3 [29], ResNet18 [30], ResNeXt101 [31] and ResNeXt152 [31]) with three datasets (Fashion MNIST [32], CIFAR-10 [33], and ImageNet [34]) as our benchmarks. Experimental results demonstrate that (1) Canary guarantees training convergence and provides robust scalability in both CPU- and GPU-equipped clusters, (2) Canary reduces traffic volume, by up to 56.28 percent less than the BML on PyTorch, (3) Canary effectively reduces training convergence time, by up to $1.61\times$, $2.28\times$ and $2.84\times$ faster than BML on PyTorch [35], Ako on PyTorch and PS on TensorFlow [36], respectively, (4) Canary can accelerate the convergence speed, while not degrading the training quality or bringing extra computational overhead, and (5) the gradient sketch method inside Canary can apply to existing learning frameworks and further improve the training efficiency. Our main contributions are summarized as follows:

- We reveal the potential communication capacity of multi-interface network to deploy distributed DL tasks and fully utilize the available bandwidth resources to accelerate parameter synchronization in a decentralized manner, while reducing communicational overhead and utilizing computational capacity.
- We conduct an algorithm- and network-level co-design to propose the *Piece-level Gradient Exchange* (PGE) and *Multi-interface Collective Communication* (MCC) for handling parameter synchronization and traffic transmission, respectively. Specifically, we design the *gradient sketch* approach based on 8-bit quantization for traffic compression and gradient tensor partition. Also, we introduce the abstraction of *colayer* to better inspect BP stages and design the colayer-level pipelining to better overlap communication and computation. In practice,



(a) Initial state. (b) Scatter/Reduce: step 1. (c) Scatter/Reduce: step 2. (d) Allgather: step 1. (e) Allgather: step 2 (done).

Fig. 2. The workflow of Ring-AllReduce adopted in Uber Horovod [37].

we elaborate two *Gradient Exchange Index* (GEI) data structures to make our algorithm online applicable.

- We implement a prototype system called Canary based on PyTorch-1.4.0 and will open source of it.[1] Canary provides uniform programming interfaces, so that developers and researchers can easily conduct decentralized DL training with little modification on their codes. Experimental results of a 34-node cluster demonstrate that Canary performs well in training generalization, scalability, communication overhead and convergence efficiency with baseline comparison.

The rest of paper is organized as follows. We introduce the background and related work in Section 2. Then, we discuss the observations motivating our research in Section 3. Detailed analysis of Canary design and implementation are given in Sections 4 and 5, respectively. We evaluate Canary in Section 6 and draw the conclusion in Section 7.

2 Background and Related Work

2.1 Machine Coordination Manner

Communication overhead is a crucial issue in data-parallel DL [16]. Most data-parallel DL frameworks coordinate machines mainly in two manners, i.e., centralized [2], [3], [5] and decentralized [12], [14] algorithms. As shown in Fig. 1a, a centralized coordination adopts a logical root to store and share parameters, so as to ensure consistency-control across the cluster. The typical PS paradigm belongs to this manner. Meanwhile, as shown in Fig. 1b, a decentralized coordination is usually implemented based on the *peer-to-peer* (P2P) communication that does not require a central node. The centralized coordination is easy to deploy, while the central root may incur bandwidth scarcity and suffer from high communication cost. In contrast, the decentralized coordination eliminates the network bottleneck from central nodes while may yield higher communication complexity to deploy underlying network topology.

2.2 Distributed Communication Mechanism

Recently, the communication mechanism based on the decentralized coordination has shown great advantages for distributed DL training acceleration. A pertinent case is the network-optimal Ring-AllReduce [38] algorithm proposed by the Baidu Silicon Valley Artificial Intelligence Lab (SVAIL), which is employed to better saturate the available bandwidth and reduce the communication overhead in High Performance Computing (HPC). We give an example in Fig. 2 to illustrate the rationale of the Ring-AllReduce algorithm. Briefly, the algorithm contains two stages, each of which is finished after transferring the tensor chunks to the neighbour peers in $2 \times (N-1)$ times (N is the number of workers). The Ring-AllReduce algorithm is easy to implement by using the collective communication abstraction based on Message Passing Interface (MPI) [39], such as the Open MPI [40] and NVIDIA NCCL [41] implementation. The Uber leverages the power of Ring-AllReduce to build the decentralized Horovod [37] framework to efficiently handle inter-GPU communication. Moreover, the widely-used industrial systems, such as PyTorch [42] and TensorFlow [36], have also adopted the NCCL techniques to accelerate the parallel communication for distributed DL training.

2.3 Parameter Synchronization Pace

In data-parallel DL training, communication of parameter synchronization will be conducted once the derivative (i.e., gradient) computation of layer weights is done. Previous researches mainly formulate this procedure via the distributed *stochastic gradient descent* (SGD) algorithm, which can be implemented in both synchronous or asynchronous schemes. *Bulk Synchronous Parallel* (BSP) [2], [14], [18] is a typical synchronous scheme, which provides stable consistency-control and guarantees stable training convergence. However, BSP may lead to time waste due to the waiting for the slowest worker, i.e., the *straggler* [1], [23]. On the contrary, *Asynchronous Parallel* (ASP) [36], [43], [44], a typical asynchronous scheme, conquers this issue by simply removing the enforced barrier synchronization at the end of each iteration. However, the *staleness* [5] in ASP may incur delay error and thus degrades the convergence efficiency. Additionally, *Stale Synchronous Parallel* (SSP) [12], [19], [20], [21], a hybrid scheme of BSP and ASP, makes a trade-off between iteration speed and convergence accuracy by introducing a delay-bounded threshold to restrict the iteration staleness and time waste among workers.

2.4 Data Quantization

Data quantization is a promising method to reduce training cost of DL tasks, where the original full-precision data are transferred into low-precision format represented by less bit width. Therefore, quantization can be used in inference acceleration and model simplification. For example, Mellempudi *et al.* [45] improved the inference efficiency by quantizing the weights and activations into INT8 values, while requiring a pre-trained model with relatively high precision. Besides, Cai *et al.* [46] aimed at quantizing the activation functions (e.g., ReLU) by using a half-wave rectifier to address the data mismatching in FP and BP approximation. Different from these algorithms, our target is to

1. https://github.com/kimihe/Canary.

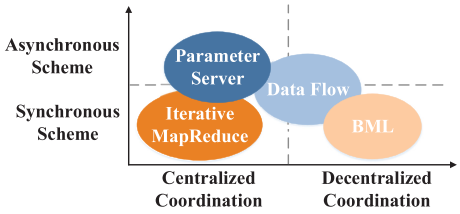


Fig. 3. Four typical architectures of data-parallel DL.

accelerate the distributed DL training speed by addressing the communication bottleneck in gradient exchanging.

2.5 Anatomy of Existing Architectures

As shown in Fig. 3, we compare existing data-parallel DL architectures of four typical types: Iterative MapReduce (e.g., Spark [47] and MLlib [48]), PS (e.g., Poseidon [2] and Adam [49]), BML [14] and Data Flow (e.g., Ako [12] and TensorFlow [36]). From the perspective of training efficiency, we take the architectures of PS and BML as case studies.

PS. To handle large-scale parallel DL, most existing work often resorts to the PS architecture. Although PS provides a simple paradigm for system implementation, the logical central servers (corresponding to workers) may face with severe communication overhead along with the scale increment of clusters, especially when bandwidth is limited. In spite of the substitute of faster network devices, such as Infiniband [50] or high-speed Ethernet, communication may still be the bottleneck [2], [12], [14].

To give a clear demonstration of network overhead, we compare the average per-synchronization computation and communication time under different models and datasets in Fig. 4. Note that the DL training tasks follow the BSP scheme and use the testbed configuration of three groups in Section 6.1.1. We can observe that the training performance suffers from non-negligible communication overhead, especially with the increase of the worker number and the model size. In addition, the communication time dominates the time cost in one synchronization when the cluster owns more powerful computational capacity. For example, in the configuration of group (1), the cluster is based on NVIDIA Tesla P100 GPUs, where the per-synchronization communication time is much longer than the computation time. This phenomenon indicates that the network overhead is a crucial issue impacting the distributed training efficiency of large models.

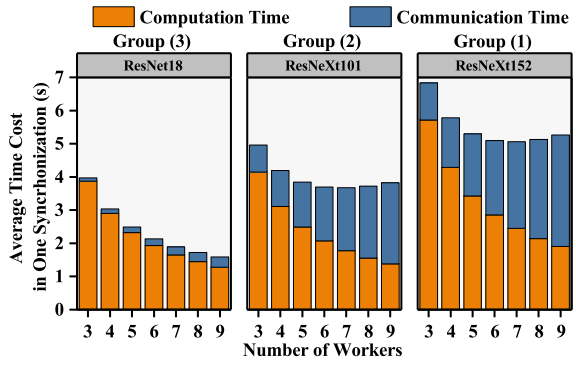
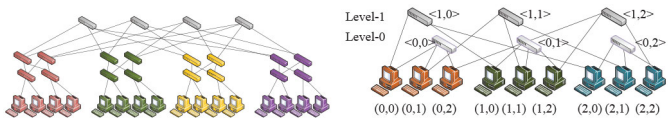


Fig. 4. The average computation (orange) and communication (dark blue) time of one synchronization in the PS cluster, with 200 global batch size and 10 Gbps Ethernet.



(a) FatTree. (b) BCube.

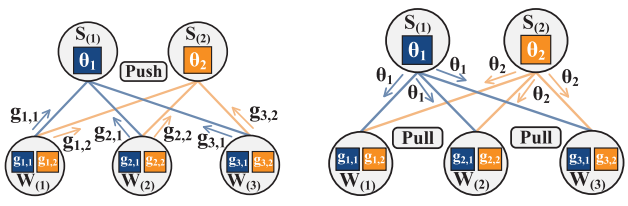
Fig. 5. Network topology.

More seriously, PS-based systems are often deployed on top of FatTree [13] network, which cannot well match the communication pattern of parallel DL training [14]. Fig. 5a describes the topology of FatTree network, where traffic flows need to pass through multiple hops of switches between two machines.

Besides, we find that *bandwidth utilization is imbalanced between workers and servers in a commodity PS architecture*. Fig. 6 shows a pertinent PS architecture, where we can observe that there are plenty of *push* (in Fig. 6a) and *pull* (in Fig. 6b) operations between servers and workers to maintain parameter consistency, where the pressure of network traffic is imbalance on servers and workers. As the number of workers is usually much more than that of servers, the servers are vulnerable to be network bottleneck when servers and workers are equipped with similar NICs in commodity data center networks. Obviously, this communication pattern will degrade the parameter synchronization efficiency and decrease bandwidth utilization.

We also conduct preliminary experiments to evidence this phenomenon. As shown in Fig. 7, we measure the real-time traffic per second on both workers and servers to show bandwidth utilization. In most cases, we can observe that the bandwidth utilization on a worker (up to 25.69 percent) is far lower than that on a server (up to 79.84 percent). Moreover, when a worker is conducting derivative computation instead of parameter synchronization, the bandwidth utilization will be even low (about 11.21 percent). It turns out that workers often waste idle bandwidth while servers may suffer from severe traffic overhead and become the bottleneck of communication efficiency. This phenomenon reveals that commodity PS on FatTree cannot well match the traffic pattern of parallel DL training and motivates us to investigate alternative networks supporting decentralized coordination.

BML. The recently presented BML architecture based on BCube [17] network addresses above issues and illustrates a promising paradigm to reduce the communication time of distributed DL systems. As shown in Fig. 5b, the recursive cube (represented as $< *, * >$) topology inside BCube inherently supports the frequent data aggregation and broadcast in data-parallel DL. Moreover, the level-based network interfaces provide more available bandwidth with a



(a) Push. (b) Pull.

Fig. 6. Centralized PS paradigm.

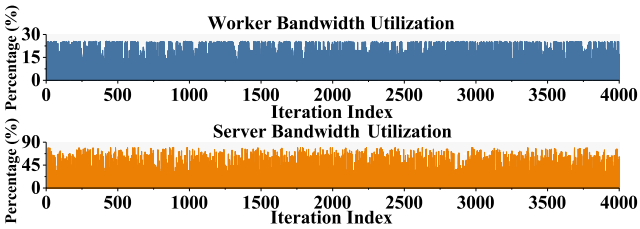


Fig. 7. We monitor real-time bandwidth utilization in PS architecture based on FatTree with 3 server and 18 workers, using VGG-19 model with CIFAR-10 dataset.

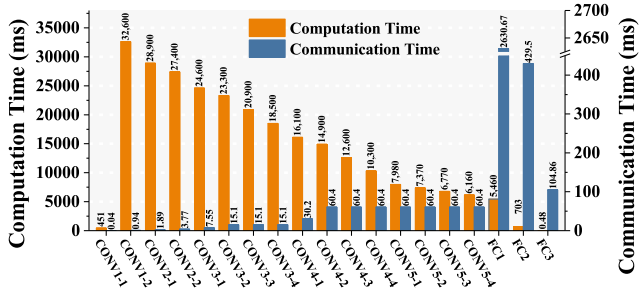


Fig. 8. Communication and computation time of different layers in 10 Gbps Ethernet. We conduct the training of image classification tasks on VGG-19 model with CIFAR-10 dataset using 32 batch size.

lower cost of switches. The direct connections inside each cube eliminate the communication complexity between different machines. These topology properties make the BCube network superior to deploy parallel DL systems over the FatTree network. However, the BSP synchronization scheme in BML may incur potential stragglers, which will slow down the training speed and degrade the convergence efficiency. The power behind BML can be summarized as two key aspects: (1) exploiting multiple NICs of each machine to reduce data transmission time, and (2) adopting decentralized gradient synchronization following the programming paradigm of Message Passing Interface (MPI) [39]. We will further discuss these two insights in Section 3.

3 Observation

We intend to design a new decentralized parallel DL system in multi-interaface network to accelerate gradient synchronization without introducing communication-level bottleneck. Four key observations guide the design of our system.

Observation 1. *The multiple interfaces of a commodity server can be used to reduce the completion time of gradient exchange.*

In a modern data center, machines with two or more ports are commonly deployed and it is affordable to conduct DL training in a cluster comprised of these machines [15]. It is natural that the inherent communication capacity of multi-interface machines can be further exploited to accelerate the communication progress. Note that few previous work has realized the promising power in this kind of network environment. A pertinent case is the recently successful BML [14] algorithm, which effectively reduces the gradient synchronization time by collaborating with a specific underlying topology called BCube [17] network. The essence of its superiority is that BML fully exploits the extra communication capacity of multiple NICs, i.e., making gradient transmission in parallel. It turns out that the success of BML relies on the specific support of BCube network. However, not all the clusters for DL training can be easily deployed on BCube topology as the change of underlying topology requires extra hardware cost. Besides, reconstructing the code to make existing DL applications compatible with BCube network is also time-consuming. Actually, many architectures have been deployed on other decentralized manner, such as Tree-based [2], [3], [11] and Mesh-based [12], [24] networks, where the machines also support multiple interfaces. This inspires us to comprehensively study *how to fully exploit the communication capacity of a general multi-interface network and accelerate the gradient exchange progress during DL training.*

Observation 2. *The gradient values of a neural network layer can be compressed with slight precision loss while effectively reducing communication traffic.*

Apart from fully utilizing the available bandwidth, another method to accelerate communication progress is reducing the network traffic, i.e., the volume of the gradients need to be transmitted. A natural way to realize this target is conducting gradient compression. Different from existing approaches focusing on the compression of the entire gradient tensor, we find that *handling compression from the perspective of neural network layers instead of the whole model can further improve the compression efficiency while guaranteeing slight precision loss of gradient values*. There are two points covering our concerns: (1) the communication and computation patterns in different layers, and (2) the distribution of gradient values in different layers.

As to the first point, the gradients of a neural network model are generated during the *backward propagation* (BP) stage following the sequence from the last layer of output to the first layer of input. We use the training of image classification tasks running in the convolutional neural networks (CNNs) as an example. Layers inside these networks can be briefly classified into three categories: convolutional (CONV) layers, fully connected (FC) layers and sampling layers (e.g., max-pooling and avg-pooling). We will inspect the CONV layers and FC layers since sampling layers do not involve computing derivatives (i.e., the gradients) and do not contain any parameters. As described in Fig. 8, we decompose the 19 layers (16 CONV layers and 3 FC layers) of VGG-19 model and find that the layers hold distinct diversity of time consumed by computation and communication. We summarize their characteristics in Table 1 and observe that a CONV layer often yields much more Float-point (32 bits) operations but fewer parameters, while an FC layer holds the inverse feature. We can observe that CONV layers require more computation time (occupying up to 91.37 percent) and FC layers dominate the communication time (occupying up to 86.35 percent). As a result, we need to carefully take the communication and computation patterns into consideration when design the compression approaches as the compression itself is also a computation-consuming operation that we should not introduce too much extra overhead.

Moreover, we select 8 representative layers inside VGG-19 and compare their gradient distribution in Fig. 9. We can observe that the gradient values of these layers follow a *sparse* and *non-uniform* distribution, where the values

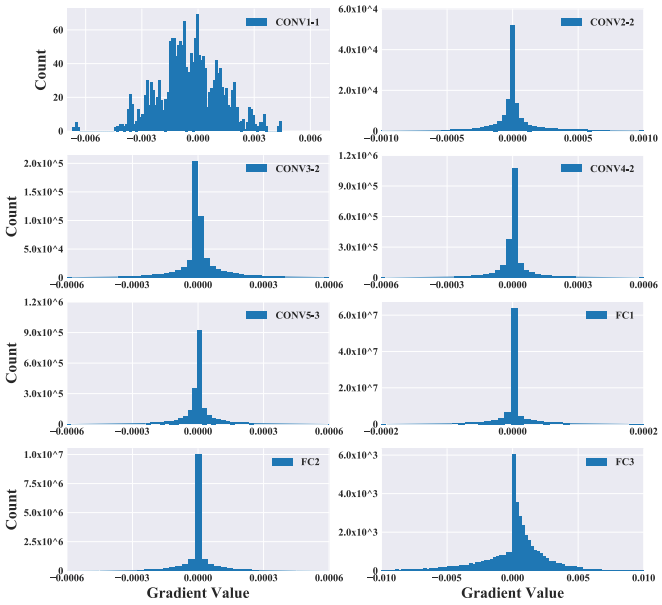


Fig. 9. Distribution of gradient values in different layers of VGG-19 model with CIFAR-10 dataset.

near zero dominate the gradient tensors. Besides, different layers hold different gradient distributions. For example, the distribution curve of CONV1 layer varies from that of FC1 layer, where the range of gradient value (X-axis) and the frequency magnitudes (Y-axis) are widely different. The deeper the layer is, the more extreme the non-uniform distribution will be. Due to the spare and non-uniform property of the gradient distribution, we cannot use traditional quantification technologies for compression, such as mapping float-point values to integers [51], since this method requires uniform data. Fortunately, with the insight of layer diversity, we can use the quantile sketch technologies [52], [53] into gradient compression, which is suitable in non-uniform scenario without losing too much gradient precision. Another important advantage is this technology primely supports our inspiration about partitioning gradient tensor into pieces (see *Observation 3*). The details of using quantile sketch to compress gradients will be discussed in Section 4.3.1.

Observation 3. *The gradient tensor can be partitioned into pieces and aggregated afterwards to fully exploit the power of parallelism and pipelining*.

As aforementioned in *Observation 1*, an effectively way to accelerate communication progress is to fully utilize the capacity of multi-interface. This requires transmitting the gradient tensor through different links simultaneously. We find that a large gradient tensor can be partitioned into a number of small pieces, so that the pieces can be transmitted via multiple interfaces in parallel. Meanwhile, pieces belonging to different parts of the gradient tensor can be aggregated partially and merged into one, i.e., following the *reduce* operation. Note that this kind of operation further decreases the gradient volume that needs to be transmitted. We use Fig. 10 to illustrate this procedure with two stages (scatter and gather), where θ and g represent global parameters and local gradients, respectively. We take worker $W_{(1)}$ as an example. In the first stage of scatter, $W_{(1)}$ partitions its

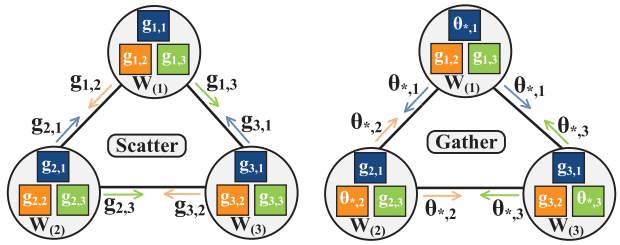
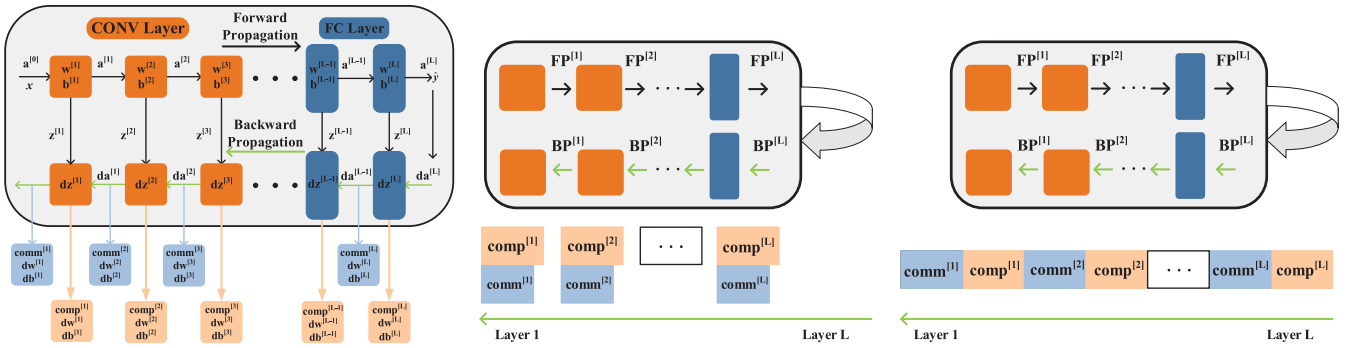


Fig. 10. Decentralized P2P paradigm.

local gradients into three pieces ($g_{1,1}$, $g_{1,2}$ and $g_{1,3}$) and sends two pieces to other two workers ($g_{1,2}$ to $W_{(2)}$ and $g_{1,3}$ to $W_{(3)}$). At the same time, the other two workers do the similar operations. As a result, $W_{(1)}$ receives two gradient pieces from other two workers ($g_{2,1}$ from $W_{(2)}$ and $g_{3,1}$ from $W_{(3)}$). With its local piece $g_{1,1}$, $W_{(1)}$ can merge these three pieces into one (i.e., $\theta_{*,1} \leftarrow g_{1,1} + g_{2,1} + g_{3,1}$), containing the computation results of other workers. The same operation is done on W_2 and W_3. In the second stage of gather, each worker holds different parts of the latest model parameters and these partial parameters can be directly broadcasted to other workers (e.g., $W_{(1)}$ broadcasts $\theta_{*,1}$ to $W_{(2)}$ and $W_{(3)}$). Eventually, all the three workers own the latest full parameters (i.e., $\theta_{*,*} \leftarrow \theta_{*,1} + \theta_{*,2} + \theta_{*,3}$) and the gradient exchange procedure is finished. Still as to $W_{(1)}$, we can observe that the key of successfully decreasing gradient volume in this example is the *reduce* operation that merges three gradient pieces into one in the first stage (i.e., $\theta_{*,1} \leftarrow g_{1,1} + g_{2,1} + g_{3,1}$ on $W_{(1)}$). Consequently, we intend to further exploit the power of gradient partition and transmission parallelism in multi-interface network.

Moreover, as shown in Fig. 11a, each iteration within a DL training process can be represented by two stages, i.e., *forward propagation* (FP) and *backward propagation* (BP). To conduct distributed DL training in a data-parallel manner, we need to exchange parameters (i.e., the gradient matrices) among different workers and promptly synchronize them to avoid introducing delay errors during the BP stage. In each layer, the computation operations for calculating gradients are independent of the communication operations for parameter synchronization among workers. Consequently, it is natural to overlap the computation and communication by adopting the *pipelining* technology. However, it is worth noting that the magnitude relationship between communication time and computation time also impacts the efficiency of pipelining in practice. As shown in Fig. 11b, to make the best use of pipelining among layers, the communication time should not be longer than computation time. It turns out that only when communication time is shorter than computation time, can pipelining effectively accelerate the entire BP stage, compared with the traditional serialization scheme in Fig. 11c. Actually, some state-of-the-art researches, e.g., Bytedance's BytePS [54], also shares some similar observations of tensor partition and communication re-scheduling. However, their design did not consider the property of the underlying network topology mentioned in Section 2.5, which significantly impacts the communication efficiency. Therefore, we intend to jointly consider the tensor partition and network



(a) Training process with forward propagation and backward propagation. (b) Backward propagation overlapping computation communication. (c) Backward propagation pipelining: serialization: operating computation and communication successively.

Fig. 11. A typical DL training contains two stages, i.e., forward propagation (FP) and backward propagation (BP). Note that backward propagation can be designed in two schemes, i.e., serialization and pipelining.

topology (e.g., FatTree and BCube) to fully exploit the power of data parallelism and pipelining. We find that the gradient partition and aggregation operations can help us reach this goal.

Observation 4. *The common decentralized synchronization algorithms can be abstracted into uniform programming interfaces based on collective communication.*

With the example mentioned in Fig. 10, we find that the essence of decentralized gradient exchange is closely related to the gradient partition and aggregation operations. This property motivates us to abstract the semantics of these operations. To make the semantics suitable for most decentralized architectures, we analyse three typical network topologies (Tree-based, Mesh-based and BCube topology) and abstract two types of stages, i.e., (1) *scatter/gather/allgather* and (2) *reduce/broadcast/allreduce*, which are refined from Message Passing Interface (MPI) [39].

Summary. These four observations motivate us to design a new decentralized DL architecture that partitions gradient into pieces via quantile sketch and conducts piece-level gradient exchange through multi-interface in parallel, so as to reduce the network traffic and accelerate the communication progress. Further, we will discuss the logical design of our system following the thought of the first three observations in Section 4 and present the implementation details corresponding to the last observation in Section 5.

4 Canary DESIGN

4.1 Definition of Terms and Notations

In order to give a clear explanation, we list the terms and notations used in our paper.

- $_{<i>}$: The index (subscript) of an iteration during DL training. Note that iteration index starts from 1.
- $_{(i)}$: The index (subscript) of a worker, e.g., $W_{(i)}$ is the i-th worker. Note that worker index starts from 1.
- $^{[i]}$: The index (superscript) of a colayer, e.g., $c^{[i]}$ is the i-th colayer. Note that colayer index starts from 0.
- $c^{[i]}_{(i)}$: The colayer $c^{[i]}$ on worker $W_{(i)}$.
- $< i, j >$: The index of a cube in multi-interface network.
- $\tau[i]$: The index of a thread.
- $< \tau[i], j >$: The index of a gradient piece.
- $< \tau[i], j > W_{(i,j)}$: The gradient piece with index of $< \tau[i], j >$ on worker $W_{(i,j)}$.
- $\boldsymbol{r}$: The GEI row vector reflecting the status of gradient piece exchange among different workers.
- $\mathbf{Q}$: The GEI matrix reflecting the progress of gradient piece exchange in time sequence with training iteration.
- ζ : The row number of $\mathbf{Q}$ reflecting the latest index of training iteration.
- s : The threshold of iteration staleness.
- p : The threshold of colayer proportion.

4.2 Overview

Our goal is to conduct a high-performance algorithm and network co-design of computation and communication stages for parallel DL training. We intend to illuminate the overview of our system from two prospectives: (1) decentralized gradient exchange in an asynchronous manner and (2) data transmission collaborated with underlying network topology. The former refers to how to elaborate an efficient parameter synchronization mechanism with robust training convergence and cluster scalability. The latter involves how to adapt our synchronization mechanism to the distributed network environment with low communication complexity and low bandwidth/link waste. We address these two prospectives with solutions *Piece-level Gradient Exchange* (PGE) in Section 4.3 and *Multi-interface Collective Communication* (MCC) in Section 4.4, respectively.

4.3 Piece-Level Gradient Exchange

4.3.1 Gradient Sketch

While the machines inside a multi-interface network own more available bandwidth, the huge traffic amount is still a crucial issue. In a commodity data center, the communication time may be unacceptably long when the available bandwidth is shared by multiple communication-intensive tasks. Consequently, apart from providing more available bandwidth, reducing the volume of network traffic is also important. To achieve this target, we compress gradient by using quantile sketch [52], [53] and we call this method as *gradient sketch*. In particular, we follow the gist of relaxed

quantization [55] and DataSketches [56] for implementation convenience. Recall that the gradient distribution varies among different layers, we take this property into consideration when using gradient sketch. The core idea of gradient sketch is to map the original large gradient tensor into a compressed array with smaller element size. Each element can be regarded as a category with a number of gradient values and we can use this element to approximately represent all the gradient values belonging to this category. This is similar to the principle of unsupervised clustering in part. The trade-off behind gradient sketch is that *we slightly degrade the precision of gradient value to obtain a prominent size decrease of gradient tensor*. It is obvious that the estimated precision is related to the category number we transferred. In practice, the category number will not exceed 256 [52], so that we can use one byte (i.e., 8 bits) to encode the index of all the categories. The gist of gradient sketch is similar to the 8-bit data quantization that represents the full-precision *32-bit floating-point* (FP32) numbers by the low-precision *8-bit fixed-point* (INT8) numbers. Meanwhile, the tensor elements are transferred from the wide-and-sharp distribution into the small-range uniform distribution. This kind of value mapping is the key to alleviate the computational pressure of matrix operations and reduce the communication traffic of gradient exchanging. Therefore, we can implement the gradient sketch method by quantizing the gradients into INT8-based data format in the BP stage. The workflow of gradient sketch is described in Fig. 12, containing three steps:

Step 1: Seek Quantiles to Split the Gradient Tensor. The original gradient tensor can be unfolded into a row vector, where the values follow the non-uniform distribution. We seek the quantiles of this row vector and split the entire vector into a number of *buckets* (i.e., categories), each of which is determined by a start quantile and an end quantile [55]. The gradient values located in the same domain will be put into the same bucket. The principle to determine suitable quantiles is that the number of gradients belonging to different buckets should be almost the same, i.e., the bucket-level gradient values follow the uniform distribution.

Step 2: Use the Mean of Start and End Quantiles to Approximate the Bucket. We calculate the mean of the start and end quantiles. All the gradient values within this bucket can be represented by this mean. That is to say, we use the mean value to approximate the entire bucket, so as to make a trade-off between gradient precision and tensor volume.

Step 3: Encode the Index of Bucket Array With Binary. As the original tensor is simplified into an array of bucket index, we can directly encode the index by using binary encoding.

More precisely, we will discuss how to use gradient sketch in different layers of a neural network. Here are two cases:

Case 1: Gradient Sketch in FC Layers. We can observe that the number of the buckets determined by the quantiles in Step 1 closely impacts the approximate precision of gradient values, we need to restrict the bucket number within an acceptable range. If too many buckets are splitted, the volume of gradient tensor will not effectively be reduced. On the contrary, the precision loss will be high when we split too few buckets. As we intend to use one byte to encode the bucket index, the maximum of bucket index is 256. This order of magnitude is enough for gradient sketch in most layers.

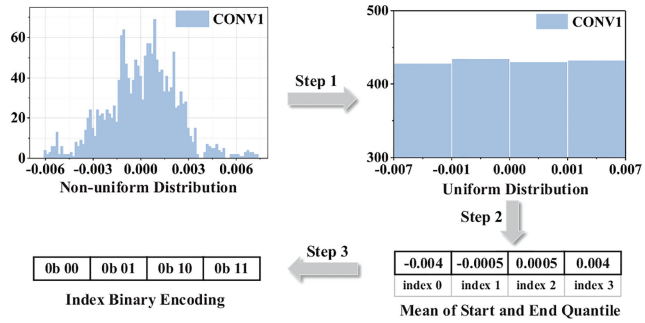


Fig. 12. The workflow of *gradient sketch* with four buckets.

However, as we mentioned in Fig. 9, the gradient values of FC layers are in a wide range with plenty of near-zero numbers, where large precision loss will be introduced even by splitting the gradient tensor into 256 buckets.

In this case, we need to reduce the scale of gradient tensor before conducting sketch.

To conquer this challenge, we introduce the matrix factorization method (i.e., *singular value decomposition* [57]) to represent the gradient tensor $\triangledown a(w, b)$ as the matrix multiplication by a few matrices or vectors. As a result, we can reduce the scale of the gradient from a huge tensor to a few factorized matrices. With this pretreatment on gradient tensors, we can operate gradient sketch on these vectors.

Case 2: Gradient Sketch in CONV Layers. The scale of gradient tensors in CONV layers (e.g., CONV 1 and CONV 2) is much smaller than that in FC layers. In this case, we can directly sort the gradient tensor and search the quantiles by counting the value number, so as to simplify the quantile calculation. This simplification can further reduce the computation time in some shallow CONV layers.

Summary. The above three steps inside gradient sketch help us transfer the original float-point gradient tensor into a small number of buckets (integer array), where the gradient values inside the same bucket are estimated by a single integer with binary encoding. Therefore, we just need to *transmit the buckets instead of the original gradient tensor*. This is the key why gradient sketch can effectively reduce the traffic volume. Note that the gradient sketch method is a special case of the 8-bit parameter quantization for gradient tensors, i.e., using small-size INT8 data to represent the gradient values, rather than using the full-precision FP32 data type [55]. The difference is that we can use fewer and flexible bits (instead of the fixed 8 bits) to compress the gradient tensors, according to the property of different layers.

Apart from the traffic compression, the buckets generated by gradient sketch conforms to our idea that partitioning gradient into pieces and conducting piece-level gradient exchange through multiple interfaces in parallel.

4.3.2 Colayer Bundling

Recall the difference about using gradient sketch in CONV and FC layers, this detailed pretreatment derives from our new abstraction called *colayer*. Colayer is introduced to better handle the granularity of gradient sketch and pipelining during BP stage. A colayer is defined as a set of layers with

analogous properties in terms of computation and communication. We can reorganize an entire neural network model into several colayers. Given the VGG-19 model with configuration described in Fig. 8, we can arrange the layers following the principle mentioned in Section 4.3.1. For example, we can bundle FC3 to FC1, CONV5-4 to CONV5-1, CONV4-4 to CONV4-1, CONV3-4 to CONV1-1 as four colayers, denoted as $c^{[0]}$, $c^{[1]}$, $c^{[2]}$ and $c^{[3]}$, respectively. Note that the re-organization criterion and the number of colayers depend on gradient distribution and parameter dimension. Therefore, the colayer number changes during DL training. In general, we mark each colayer with an index $c^{[i]}$, which is used to trace the entire gradient exchange progress. The detailed function of colayer index will be illustrated in Section 5.2. Introducing the abstraction of colayer brings the following two advantages.

Advantage 1: Computation Conservation. As shown in Fig. 11a, we can avoid recomputing the part of derivatives during BP by better reusing the cached object z inside activation function a. Note that these last two CONV layers yield relatively fewer float-point operations compared with other CONV layers. Therefore, we bundle them together as a colayer to improve system coding efficiency when conducting gradient sketch. It is worth noting that the colayer reorganization criterion is a trial-and-error exploration, which can be regarded as a hyper-parameter search. Consequently, the aforementioned colayer reorganization is the empirical tailoring of hyper-parameter setting from our experiments.

Advantage 2: Pipelining Acceleration. As the traffic volume is reduced by using gradient sketch, the communication time of a layer during BP stage is shorter than the computation time in most cases. Besides, by adopting the colayer-level re-organization, we can ensure that the communication time is prominently shorter than the computation time. This provides the opportunity to introduce pipelining technology to accelerate the BP stage. Note that using pipelining is not new. The previous approach, Poseidon [2], provides the *wait-free backpropagation* to achieve a similar function. However, it conducts pipelining in an over fine-grained manner, i.e., overlaps communication and computation in each layer inside BP, regardless of inter-layer characteristics and bandwidth availability. We handle pipelining in the colayer level, where the computation and communication properties of different layers are considered. This makes the pipelining module in our system better overlap the computation and communication when handling DL training.

4.3.3 Gradient Exchange Algorithm

To build a high-performance decentralized DL system, we design our gradient exchange algorithm with the support of underlying multi-interface network. The server-centric BCube topology is a good candidate as it provides lower transmission latency and is RDMA friendly [15]. Consequently, we use the BCube network as a pertinent case to show the implementation of our algorithms. Other multi-interface networks (e.g., Tree-based and Mesh-based networks) follow the same programming interfaces as mentioned in *Observation 4*, so that our algorithm can be easily deployed on these networks. We observe that BML [14] has pioneered how to collaborate with BCube network. However, the gradient synchronization algorithm used in BML follows the BSP scheme, which may suffer from the potential stragglers because the barrier synchronization at the end of each iteration is bounded by the slowest worker [5]. More seriously, the idle time on waiting for the stragglers will be more obvious when workers own significant differences of computation capacity in the heterogeneous environment. A natural approach to address this issue is to introduce the asynchronous distributed SGD algorithm (i.e., ASP) for gradient exchange. However, this asynchronous coordination manner will incur staleness which may lead to poor convergence results. To make a trade-off between straggler and staleness, we intend to design a new gradient exchange algorithm based on SSP but in a manner that uses piece-level collective communication.

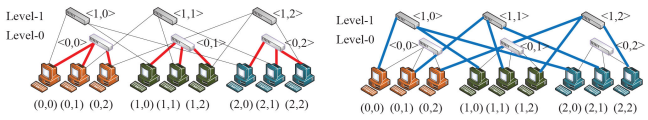
The main idea of our algorithm is to asynchronously and partially exchange gradient in the colayer granularity, using gradient sketch to conduct traffic compression and gradient partition. With the proposed *Gradient Exchange Index* (GEI) data structures (Section 5.2), we can readily monitor the gradient exchange progress of all workers. Meanwhile, we set two thresholds to restrict the delay-bounded staleness: (1) traditional *staleness* threshold s handling the synchronization strides among workers and (2) the *proportion* threshold p that controls how many coloayers on each worker should exchange corresponding gradient tensors before s is exhausted, otherwise gradient tensors that have been exchanged are marked as out-of-date and should be dropped. The details of staleness bounding are illustrated in Section 5.3.

4.4 Multi-Interface Collective Communication

With the gradient exchange algorithm controlling parameter synchronization, we need to design an efficient data transmission mechanism that adapts our algorithm to the underlying multi-interface network. We select BCube network as the candidate due to its better communication performance over other networks [15]. Our synchronization algorithm can also be easily transferred into other networks as we provide the uniform programming interfaces (Section 5).

4.4.1 BCube Topology

Following the classical definition of BCube network [14], [17], the total worker number n in BCube(α, β) can be defined as: $n = \alpha^{\beta}$, where α is the port number of each switch and β is the recursive levels of cubes $< *, * >$. We intend to partition gradient tensors into pieces and asynchronously transmit these pieces through the underlying BCube network in parallel, so as to avoid frequent traffic burst and reduce the communication overhead. To achieve this goal, we need to efficiently exploit two inherent properties of BCube network, i.e., (1) inter-cube connection and (2) multi-level interfaces of each worker. The first property ensures that workers belonging to the same cube can directly communicate with each other in a P2P manner instead of through a multi-hop route. Note that the cubes follow the recursive topology with β levels and each worker is located in β cubes at the same time. Also, the second property provides $\beta\times$ more available bandwidth over



(a) Intra-cube communication in level-0 links. (b) Inter-cube communication in level-1 links.

Fig. 13. Gradient pieces are exchanged through intra- and inter-cube communication.

TABLE 2
Synchronization Procedure of Intra- and Inter- Gradient Exchange on Worker $W_{(0,0)}$

Step	Stage	Gradient Pieces Collected on $W_{(0,0)}$
Step1	*allgather*	$< \tau[0], 0 > W_{(0,*)}; < \tau[1], 0 > W_{(*,0)}$
Step2	*allgather*	$< \tau[0], 0 > W_{(*,*)}; < \tau[1], 0 > W_{(*,*)}$
Step1	*allreduce*	$< \tau[0], * > W_{(*,*)}; < \tau[1], * > W_{(*,*)}$
Step2	*allreduce*	$< \tau[*], * > W_{(*,*)}$

commodity FatTree network and we can conduct communication of different cubes through these multi-interfaces. For handling β interfaces on a worker, we employ β threads to manage the communication on each interface. Meanwhile, we identify a gradient piece as $< \tau[*], * > W_{(*)}$, where $W_{(*)}$ and $\tau[*]$ represent the index of worker and thread, respectively. Based on this identification, we can transmit gradient pieces among workers via different interfaces simultaneously without inter-cube bandwidth competition.

4.4.2 Synchronization Algorithm

With gradient sketch, the gradient tensor is partitioned into pieces (buckets), so that we can exchange them among workers in a decentralized manner. We design a synchronization algorithm based on the programming interfaces extracted from *Observation 4* to operate this asynchronous procedure. To show the compatibility of our synchronization algorithm, we will discuss the gradient exchange procedure in the decentralized environment of BCube and Mesh networks.

BCube Network. The algorithm contains two stages, i.e., *allgather* and *allreduce*, when deploying in the BCube network. In the allgather stage, a worker collects all the gradient pieces with the same index from other workers. In the allreduce stage, a worker exchanges each fully aggregated gradient piece to other workers. Both two stages contain intra- and inter-cube communication, which can be operated via multi-level threads in parallel.

As shown in Fig. 13, we use an example based on BCube (3,2) with 9 workers to illustrate the communication procedure. These 9 workers are marked by their index, ranging from $W_{(0,0)}$ to $W_{(2,2)}$. As each worker owns 2 interfaces, we allocate 2 threads on each worker. Besides, we partition each worker's gradient into 4 $(2 \times 2 = 4)$ pieces, which are marked by using index from $< \tau[0], 0 >$ to $< \tau[1], 1 >$. Therefore, we have 36 $(9 \times 4 = 36)$ gradient pieces in total. All these pieces can be identified by using piece index from $< \tau[0], 0 > W_{(0,0)}$ to $< \tau[1], 1 > W_{(2,2)}$ uniquely. For brief, we take $W_{(0,0)}$ as an example. The entire communication procedure is summarized in Table 2. Note that both allgather and allreduce stages contain two steps. In the first step of allgather stage, $W_{(0,0)}$ collects gradient pieces of level-0 cube $< 0, 0 >$ (Fig. 13a) and level-1 cube $< 1, 0 >$ (Fig. 13b) via thread $\tau[0]$ and $\tau[1]$, respectively. After combining its local gradient pieces $< \tau[0], 0 > W_{(0,0)}$ and $< \tau[1], 0 > W_{(0,0)}$, $W_{(0,0)}$ gets $< \tau[0], 0 > W_{(0,*)}$ and $< \tau[1], 0 > W_{(*,0)}$. In the second step, $W_{(0,0)}$ exchanges the partial aggregated pieces to other workers while handles cube $< 0, 0 >$ and cube $< 1, 0 >$ via thread $\tau[1]$ and $\tau[0]$, respectively. After that, $W_{(0,0)}$ owns two fully aggregated pieces $< \tau[0], 0 > W_{(*,*)}$ and $< \tau[1], 0 > W_{(*,*)}$. Then, the synchronization comes to the allreduce stage. In the first step of allreduce stage, $W_{(0,0)}$ exchanges the two fully aggregated pieces to other workers, and operates cube $< 0, 0 >$ and cube $< 1, 0 >$ via thread $\tau[1]$ and $\tau[0]$, respectively. After this exchanging, $W_{(0,0)}$ owns fully aggregated pieces $< \tau[0], * > W_{(*,*)}$ and $< \tau[1], * > W_{(*,*)}$. Finally, in the second step of allreduce stage, $W_{(0,0)}$ can simply combine these two pieces to form the final complete gradient tensor.

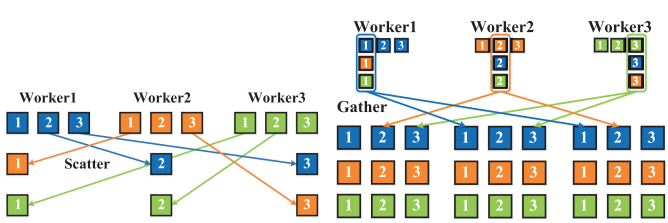
Different from BML that conducts above procedure using BSP, we handle this in an asynchronous scheme. In our synchronization algorithm, each worker independently operates gradient exchange without a synchronized barrier at the end of each iteration. Consequently, other workers may be still under computation when $W_{(0,0)}$ is operating gradient exchange. Each worker will conduct gradient exchange when it has finished the derivative computation. In a macro view, the interleaved communication among workers is operated through the BCube network in parallel. To avoid extra traffic overhead, we stipulate that a worker will send an empty packet when it is still under computation or has exchanged current gradient in prior communication. In programming practice, we implement this role with the support of two GEI data structures (Section 5.2).

Mesh Network. As to the case in the Mesh network, our algorithm can be further simplified into one stage of *allgather* with two steps. We use Fig. 14 to better illustrate the gradient exchange procedure. In the first step, each worker scatters the gradient pieces to the other $N-1$ workers (assuming there are N workers in total) by matching the piece index to the worker index. Thus, each worker owns the partial-aggregated gradient tensor (i.e., the 3 pieces in Fig. 14a). In the second step, each worker gathers the partial-aggregated tensors from other $N-1$ workers and finally forms the fully aggregated gradient tensors (i.e., the 9 pieces in Fig. 14b). The communication of these two steps can be operated in parallel by using the Open MPI [40] or NVIDIA NCCL [41] library.

5 Implementation

5.1 System Architecture and APIs

We implement Canary on PyTorch-1.4.0 [35] with CUDA-10.1 [58] and NVIDIA NCCL [41]. With the programming-



(a) Step 1: scatter gradient pieces. (b) Step 2: gather partial-aggregated tensors.

Fig. 14. The workflow of synchronization algorithm in Mesh network.

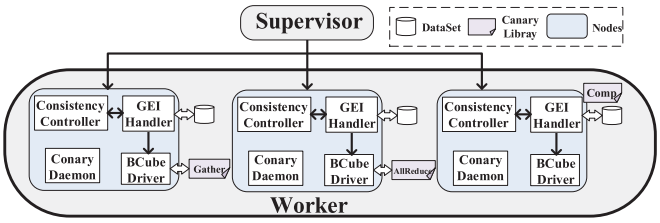


Fig. 15. Architecture overview of Canary.

friendly PyTorch, we utilize the Python-level libraries to build the model training functions. Other functions related to the distributed communications are developed via the hybrid coding with C++, using the Open MPI [40] tools and PyTorch distributed communication package [59]. Note that the CUDA and NCCL toolkits will be invoked when GPUs are available for training. As shown in Fig. 15, Canary follows a decentralized structure, where a cluster manager called *supervisor* monitors the entire cluster and workers communicate to each other in a P2P manner without the message forwarding by the supervisor. On each worker, a Canary daemon handles the DL training process and machine execution status. The GEI handler records the intermediate training data during BP stages and collaborates with the consistency controller to restrict staleness in an acceptable bound. All the gradient tensors are partitioned under the control of GEI handler and are transmitted via BCube driver. Note that the BCube driver abstracts the underlying network topology to decouple the traffic transmission with upper synchronization algorithm. Moreover, we develop the Canary API (in Table 3) in Python for practical implementation. We can create a `CanarySession` object to interact with Canary. For example, we handle the allreduce stage as follows.

```
  cs = new CanarySession(workerId=(0,0))
  cs.all_reduce
(tensor, cubeId=<0,0>, piece_level=2)
```

With the Canary library, we can expediently conduct parallel DL training and leverage the superior of BCube network.

5.2 GEI Handler

The most crucial issue in practice is to make Canary online applicable. To achieve this goal, we elaborate two data structures, i.e., *Gradient Exchange Index* (GEI) *Row Vector* and *GEI Matrix* to handle the global coordination of parameter synchronization across the cluster. The GEI module is the core of our gradient exchange algorithm.

GEI Row Vector: $\boldsymbol{r}$.Given a worker $W_{(i)}$ and colayer $c^{[i]}$, we use $c^{[i]}_{(i)}$ to represent colayer $c^{[i]}$ on worker $W_{(i)}$.[2] With the colayer total amount k, we use the GEI row vector $\boldsymbol{r}_{(i)}$ to mark the gradient exchange progress of all colayers (from $c^{[0]}_{(i)}$ to $c^{[k-1]}_{(i)}$) on $W_{(i)}$. As shown in Fig. 16a, we illustrate $\boldsymbol{r}_{(i)}$ in the format of a row vector containing k elements, each of which records the parameter synchronization status of $c^{[i]}_{(i)}$.

2. To avoid confusion, we use subscript $_{(i)}$, superscript $^{[i]}$ and subscript $_{<i>}$ to represent the index of workers, colayers and iterations, respectively.

TABLE 3
Canary API

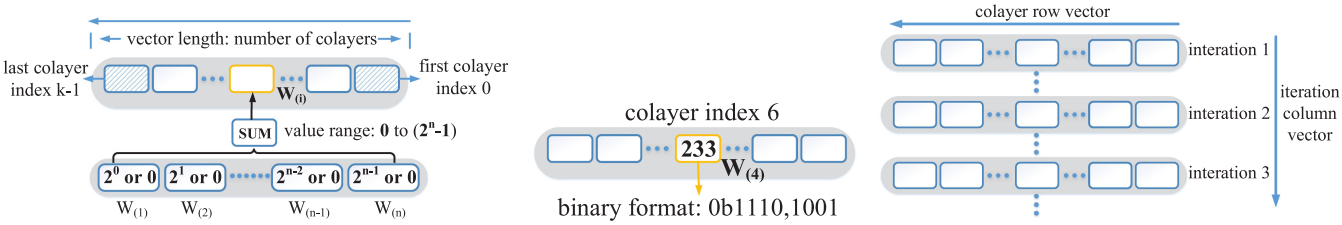
CanarySession Methods	Module
all_gather(tensor, cube, pieces)	BCube Driver
all_reduce(tensor, cubeId, piece_level)	BCube Driver
GEI_row_vector	GEI Handler
GEI_matrix	GEI Handler
calc_staleness(int, mask)	Consistency Controller
calc_proportion($\boldsymbol{r}_{(i),<i>}$)	Consistency Controller
check_staleness($s_{(i),<i>}$, threshold)	Consistency Controller
check_proportion($p_{(i),<i>}$, threshold)	Consistency Controller

Note that $\boldsymbol{r}_{(i)}$ is unfolded from the right to the left, thus the first element $r^{[0]}_{(i)}$ corresponds to the first colayer $c^{[0]}_{(i)}$ and the last element $r^{[k-1]}_{(i)}$ corresponds to the last colayer $c^{[k-1]}_{(i)}$. In a n-worker cluster, each worker needs to receive n-1 gradient tensors in total from other n-1 workers. On worker $W_{(i)}$, we encode the gradient piece index of all colayers (denoted as $c^{[*]}_{(i)}$) with the unique *integer* value[3] (i.e., $2^{(i)-1}$) to distinguish $W_{(i)}$'s all colayers $c^{[*]}_{(i)}$ from other workers'. Therefore, we can cumulate another worker's (e.g., $W_{(j)}$) encoded gradient index (e.g., index of $c^{[i]}_{(j)}$) to $r^{[i]}_{(i)}$ when $W_{(i)}$ has received gradient pieces of corresponding colayer $c^{[i]}_{(j)}$ from $W_{(j)}$. Otherwise, we cumulate 0 to $r^{[i]}_{(i)}$ to represent that $W_{(i)}$ has not received any gradient pieces from other workers. Note that the index of local gradient pieces generated by $W_{(i)}$ will also be added to $r^{[i]}_{(i)}$. Therefore, $r^{[i]}_{(i)} = 0$ indicates that $W_{(i)}$ has not exchanged any gradient pieces of colayer $c^{[i]}$ with other n-1 workers and the local gradient pieces of $c^{[i]}_{(i)}$ also has not been figured out.

We use an example to illustrate this procedure. As shown in Fig. 16b, assume that $r^{[6]}_{(4)}$ (corresponding to colayer $c^{[6]}_{(4)}$ on worker $W_{(4)}$) is 233, we can search out which workers have exchanged their gradients of $c^{[6]}$ to $W_{(4)}$ by inspecting the value of $r^{[6]}_{(4)}$ in binary format. As 233 is translated as "$0b\ 1110, 1001$", we figure out that $W_{(4)}$ has received gradient pieces belonging to $c^{[6]}$ from workers with index of 1, 4, 6, 7 and 8. Note that worker index of 4 (i.e., $W_{(4)}$) is also recorded in "$0b\ 1110, 1001$". This indicates that $W_{(4)}$ has also sent its gradient piece of $c^{[6]}$ to other n-1 workers, thus we add "$0b\ 0000, 1000$" into $r^{[6]}_{(4)}$ to record this operation.

GEI Matrix: $\mathbf{Q}$. As to $W_{(i)}$, the $\boldsymbol{r}_{(i)}$ records the status of gradient piece exchange of all colayers. However, the DL training on each worker contains a succession of iterations, each of which exchanges gradient pieces among workers during the BP stage. Therefore, we introduce the time sequence (i.e., the index of training iteration $_{<i>}$) into $\boldsymbol{r}_{(i)}$ to better describe the progress of entire distributed DL procedure. The original row vector is modified as $\boldsymbol{r}_{(i),<i>}$, where subscript $_{<i>}$ represents the iteration index. As shown in Fig. 16c, the multiple BP stages in this iterative training process can be defined as a GEI matrix $\mathbf{Q}$, each row of which is a GEI row vector $\boldsymbol{r}$. The row number ζ of $\mathbf{Q}$ represents the latest index of training iteration. In the parallel asynchronous coordination, a new $\boldsymbol{r}_{(*),<i+1>}$ will be created once any worker has stepped into a new iteration $_{<i+1>}$ from

3. In our experiments, the worker number is 34. Although n may be large in datacenter environment as the cluster contains plenty of workers, it is still feasible to allocate an integer variable of this magnitude in Python, even up to $2^{1,000}$.



(a) GEI row vector $\boldsymbol{r}_{(i)}$ on worker $W_{(i)}$.

(b) Index of $r_{(4)}^{[6]}$ is converted as binary "0*b* 1110, 1001".

(c) GEI matrix $\mathbf{Q}$ is comprised of a series of vectors under successive iterations.

Fig. 16. GEI data structures are employed to trace the gradient exchange process on each worker.

current iteration $_{<i>}$ and this latest iteration index will be spread to all workers. As a result, the row number ζ of $\mathbf{Q}$ is determined by the fastest worker in the cluster. As each worker independently maintains a local copy of $\mathbf{Q}$, the first $_{<i-1>}$ rows of $\boldsymbol{r}$ are the same on all workers when the latest iteration index is $_{<i>}$. Only the last row may be slightly delayed before the latest inter-worker communication is done. Therefore, all workers own a same $\mathbf{Q}$ when each gradient exchanging operation is completed.

Summary. With these two data structures, we can handle the iterative training process of each colayer among different workers and the asynchronously exchange gradient pieces through our underlying communication module.

5.3 Staleness Bounding

Recall that we set two thresholds (i.e., staleness s and proportion p) to restrict the delay-bounded staleness in our algorithm, so as to guarantee the training convergence efficiency in practice. As a result, how to effectively monitor these two thresholds is also important. With the GEI data structures proposed in Section 4.3.3, we can easily trace the influence of these two thresholds. Given a worker $W_{(i),<i>}$ in current iteration $_{<i>}$, we mainly consider the following three issues to monitor these two thresholds.

1. How to calculate current staleness value $s_{(i),<i>}$? Current staleness value $s_{(i),<i>}$ is defined as the gap of current iteration index between $W_{(i)}$ and the fastest worker, i.e., the difference of $_{<i>}$ and the row number ζ. As each worker independently maintains a local copy of $\mathbf{Q}$, we can readily calculate $s_{(i),<i>}$ as $s_{(i),<i>} = \zeta -_{<i>}$ (line 5 Pseudocode 1).

2. How to calculate current proportion value $p_{(i),<i>}$? We can count the number of colayers that have finished exchanging gradient pieces, so as to calculate current proportion value $p_{(i),<i>}$. We scan each element $r_{(i),<i>}^{[i]}$ in $\boldsymbol{r}_{(i),<i>}$ and inspect the binary digit of $r_{(i),<i>}^{[i]}$ to check whether $W_{(i),<i>}$ has exchanged the gradient pieces corresponding to colayer $c_{(i),<i>}^{[i]}$. We need to inspect how many $r_{(i),<i>}^{[i]}$ satisfying this condition to figure out $p_{(i),<i>}$ (lines 9–16 Pseudocode 1). Actually, we can use the *bitwise* operation to simplify this process (line 2 Pseudocode 1). Moreover, based on $p_{(i),<i>}$, it is readily to check whether $W_{(i)}$ has exchanged all the gradient tensor pieces with other workers in iteration $_{<i>}$.

3. Given two thresholds s and p, how to inspect whether the local gradient on worker $W_{(i),<i>}$ is out-of-date? With $s_{(i),<i>}$ and $p_{(i),<i>}$, we can check whether $p_{(i),<i>}$ is smaller than threshold p (line 22 Pseudocode 1) when $s_{(i),<i>}$ has exceeded threshold s (line 19 Pseudocode 1). If this condition is true, we will mark $W_{(i),<i>}$ as overtime and the gradient tensors generated in current iteration will be dropped. Then, $W_{(i),<i>}$ will directly pull the latest model parameters from the fastest worker. This bounding operation effectively avoids introducing too much delay error that may degrade the training convergence.

Pseudocode 1. Staleness Bounding

```
GEI Data Structure (Require W_(i),<i>, s, p):
1: procedure BitwiseAnd(Int r, Mask mk)
2:     return binary(r) ∧ binary(mk);                ▷ Bitwise "AND".
3: end procedure
4: procedure CalcStaleness(Int r, Mask mk)
5:     return s_(i),<i> ← ζ −<i>;                    ▷ Current staleness value
6: end procedure
7: procedure CalcProportion(GEI Row Vector r_(i),<i>)
8:     p_(i),<i> ← 0;
9:     for r^[i]_(i),<i> ∈ r_(i),<i> do
10:        if BitwiseAnd(r^[i]_(i),<i> , _(i)) == 1 then
11:            p_(i),<i> ← p_(i),<i> + 2^[i];
12:        else
13:            p_(i),<i> ← p_(i),<i> + 0;
14:        end if
15:    end for
16:    return p_(i),<i>; ▷ Current proportion value.
17: end procedure
18: procedure CheckStaleness(s_(i),<i>, Threshold s)
19:    return (s_(i),<i> ≤ s) ? 1 : 0;               ▷ Boolean return.
20: end procedure
21: procedure CheckProportion(p_(i),<i>, Threshold p)
22:    return BitwiseAnd(p_(i),<i>, p);              ▷ Boolean return.
23: end procedure
```

It is worth noting that we can switch our algorithm to BSP, ASP and traditional SSP by adjusting the threshold of staleness s and proportion p. We use $W_{(i)}$ as an example. First, we can set p as the colayer number or the length of GEI row vector $\boldsymbol{r}$ to shift our algorithm from piece-level gradient exchange to full gradient exchange. Then, we control the value of $r_{(i),<i>}^{[i]}$ to distinguish the synchronous and asynchronous coordination. As to BSP, we restrict $r_{(i),<i>}^{[i]}$ as 2^n-1 and require $W_{(i)}$ receiving all the gradient pieces from other $n-1$ workers at the end of each iteration, i.e., $s_{(i),<i>}$ is always 0. As to ASP and SSP, we can soften the restriction and permit $r_{(i),<i>}^{[i]}$ ranging from 0 to 2^n-1. More precisely, we can set $s_{(i),<i>} \leq s$ to shift into SSP while ASP has no requirements on $s_{(i),<i>}$.

6 Evaluation

We evaluate Canary mainly in the following five aspects:

1. *How does* Canary *perform in convergence?* As to image classification, Canary achieves robust training convergence and minimizes the loss function under different benchmarks, in both CPU- and GPU-equipped clusters.
2. *How does*Canary *perform in scalability?* Canary can provide stable test accuracy and good speedup of image processing speed along with the increment of worker number, ranging from the BCube(2, 2) to BCube(5, 2) cluster.
3. *How does*Canary *reduce communication overhead?* The records of real-time bandwidth utilization demonstrate that Canary requires less average bandwidth utilization and yields less traffic burst compared with the BML on PyTorch. Moreover, Canary generates less network traffic by up to 56.28 percent reduction over BML on PyTorch.
4. *How is the performance of*Canary*'s gradient sketch?* The gradient sketch method can effectively accelerate the convergence speed, while not degrading the training quality and not bringing extra computational overhead to the distributed training system.
5. *How does*Canary *improve training efficiency?* As to the convergence time and test accuracy with given time, Canary provides better performance over BML on PyTorch, Ako on PyTorch and PS on TensorFlow. Moreover, the gradient sketch method inside Canary can apply to existing learning frameworks and further improve the training efficiency.

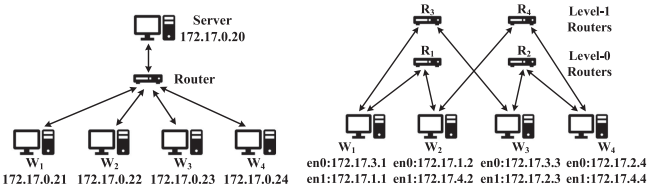
6.1 Evaluation Setting

6.1.1 Testbed

Our experiments are conducted on a 34-node cluster in Alibaba Cloud [26]. We arrange the entire cluster into three groups: (1) 16 nodes virtualized from two GN5 instances, owning 16 NVIDIA Tesla P100 GPUs with 256 GB graphic memory, 104-core vCPUs and 960 GB RAM in total; (2) 9 nodes virtualized from one GN6 instance, owning 8 NVIDIA Tesla V100 GPUs with 128 GB graphic memory, 88-core vCPUs and 256 GB RAM in total; and (3) 9 nodes virtualized from 9 C5 instances, each of which owns a 8×2.5 GHz CPU with 16 GB RAM without GPU. Note that each node enables 2 network interfaces and runs Ubuntu 16.04.2 LTS with GNU/Linux 4.8.0-36-generic kernel.

6.1.2 Network Construction

All the nodes are virtualized from the GN5 and GN6 instances with the support of Alibaba Cloud Elastic GPU Service [60]. By using the tool of Elastic Network Interfaces (ENIs) [61], we can configure multiple ENIs for each instance. To construct the network links, we deploy the instances in a 10 Gbps Ethernet local area network, where all the nodes are based on independent virtual machines, each of which owns different ENIs and private IP addresses. We construct the Tree-based and BCube(*,2) network with 4, 9, 16 and 25 workers by configuring IP address and routing table among groups. The detailed network constructions of FatTree and BCube topology are shown in Figs. 17a and 17b, respectively. As to the FatTree topology, we use the PS example based on 1 server and 4 workers to show the network construction. The server and four workers are located in the local area network, communicating with each other via the router. Their IP addresses can be easily set within the same network segment (e.g., following the rule of `172.17.0.0/24`). Meanwhile, we use the case of BCube(2,2) to illustrate the BCube construction, where each worker owns two ENIs and connects to the two-level (level-0 and level-1) routers. We identify each router and worker with a unique index. The IP address definition rule follows *"172.17.router-index.worker-index"*. For example, the two interfaces `en0` and `en1` of W_1 can be identified as `172.17.3.1` and `172.17.1.1`, respectively. Note that a worker only communicates to others whose ENIs are connected to the same routers. For example, the `en0` of W_1 can directly communicate to the `en0` of W_3 via R_3. Overall, we can establish the network links between two workers with the support of ENIs and IP address assignment.



(a) The construction of FatTree topology. (b) The construction of BCube(2,2) topology.

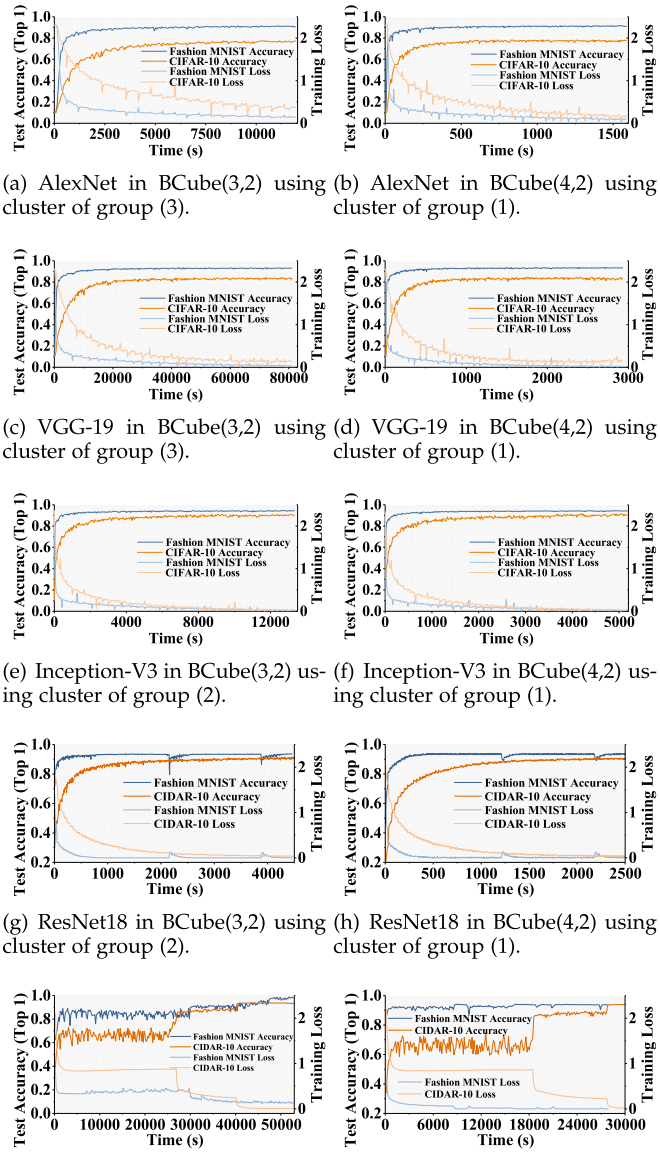
Fig. 17. The construction example of network links.

6.1.3 Workload and Benchmark

We conduct the DL training of image classification tasks based on six models (AlexNet [27], VGG19 [28], Inception-V3 [29], ResNet18 [30], ResNeXt101 [31] and ResNeXt152 [31] with three datasets (Fashion MNIST [32], CIFAR-10 [33], and ImageNet [34]). We empirically set the global batch size as 256 and the local batch size will be adjusted under different number of workers. For example, the local batch size is round to 10 when the worker number is 25. Following the guidelines from ConSGD [1], we set the initial learning rate as 0.1 and multiply it by 0.2 when every 25 percent training progress is done. Besides, we implement three baselines: BML on PyTorch [35] (BML-PT), Ako on PyTorch (Ako-PT) and PS on TensorFlow [36] (PS-TF), both of which are based on BSP. Furthermore, we apply the gradient sketch method into existing distributed DL training frameworks, e.g., Horovod [37] and BytePS [54], to inspect the improvement of image processing speed.

6.2 Convergence and Generalization

We mainly focus on the metrics of top one test accuracy and training loss with certain epochs (40 for AlexNet, VGG19, Inception, 100 for ResNet18, and 200 for ResNeXt101 and ResNeXt152). Considering the attributes of different models and datasets, we deploy the training in different groups of cluster based on BCube(3, 2) and BCube(4, 2). As shown in Fig. 18, regardless of different models and datasets, Canary can achieve stable test accuracy and minimize loss function after sufficient training epochs. For example, we conduct image classification on Inception-V3 with Fashion MNIST



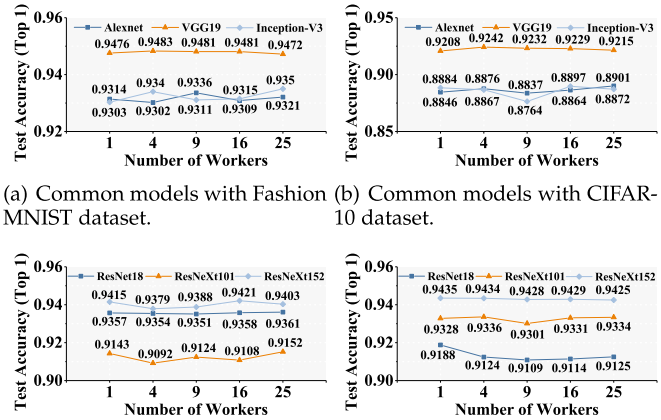
(a) AlexNet in BCube(3,2) using cluster of group (3).
(b) AlexNet in BCube(4,2) using cluster of group (1).
(c) VGG-19 in BCube(3,2) using cluster of group (3).
(d) VGG-19 in BCube(4,2) using cluster of group (1).
(e) Inception-V3 in BCube(3,2) using cluster of group (2).
(f) Inception-V3 in BCube(4,2) using cluster of group (1).
(g) ResNet18 in BCube(3,2) using cluster of group (2).
(h) ResNet18 in BCube(4,2) using cluster of group (1).
(i) ResNeXt101 in BCube(3,2) using cluster of group (2).
(j) ResNeXt152 in BCube(4,2) using cluster of group (1).

Fig. 18. Sequence diagram of training convergence in terms of top one test accuracy and training loss using different workload and testbed configurations.

and CIFAR-10 in GPU V100 (in Fig. 18e) and GPU P100 (in Fig. 18f) cluster, respectively. We observe that Canary achieves good training generalization in both BCube(3, 2) and BCube(4, 2) clusters, where the top one test accuracy is up to 94.81 percent (with 0.29 percent training loss) and 94.29 percent (with 0.75 percent training loss) on Fashion MNIST and CIFAR-10, respectively. Moreover, although V100 and P100 own analogous computational capacity, the training convergence time in V100 cluster is longer than that in P100 cluster as BCube(4, 2) can provide more communication power over BCube(3, 2). The details about Canary's scalability will be discussed in Section 6.3.

6.3 Scalability

The scalability is also a crucial metric to evaluate Canary in realistic deployment. We mainly consider three aspects, i.e., top one test accuracy (in Fig. 19), image processing speed



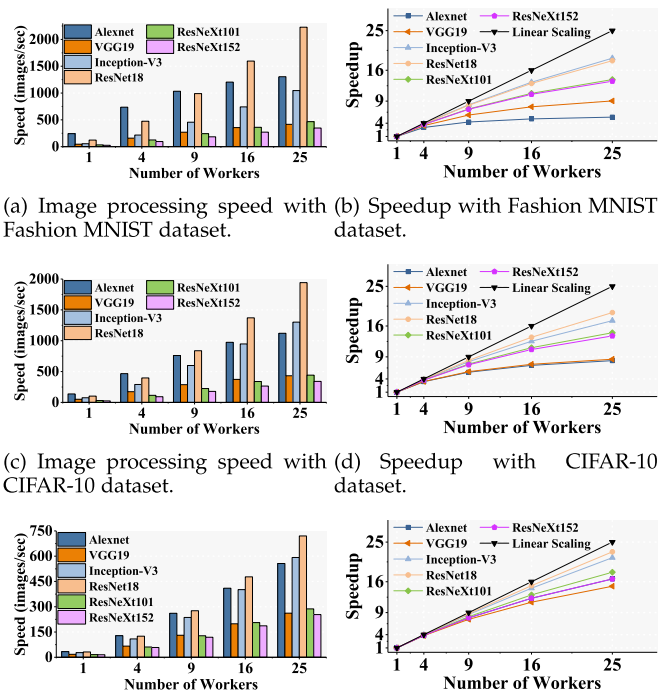
(a) Common models with Fashion MNIST dataset.
(b) Common models with CIFAR-10 dataset.

(c) ResNet-based models with Fashion MNIST dataset.
(d) ResNet-based models with CIFAR-10 dataset.

Fig. 19. Canary guarantees adequate top one test accuracy with different umber of workers.

and training speedup (in Fig. 20) with the increase of worker number. Experiments are conducted on six models with two datasets. As shown in Fig. 19, Canary can yield acceptable top one accuracy until training convergence in different cluster scales, from 4-node BCube(2, 2) to 25-node BCube(5, 2). This indicates that the staleness-bounded asynchronous gradient exchange provides Canary with good robustness while avoiding being trapped in poor local convergence. Meanwhile, we observe that the training performance does not degrade in large-scale networks, where Canary can make a good balance between consistency-control and communication overhead.

Moreover, we measure the image processing speed (images/sec) and the corresponding speedup in different



(a) Image processing speed with Fashion MNIST dataset.
(b) Speedup with Fashion MNIST dataset.
(c) Image processing speed with CIFAR-10 dataset.
(d) Speedup with CIFAR-10 dataset.
(e) Image processing speed with ImageNet dataset.
(f) Speedup with ImageNet dataset.

Fig. 20. Canary improves the image processing speed (images/sec) and achieves good speedup on six models with three datasets.

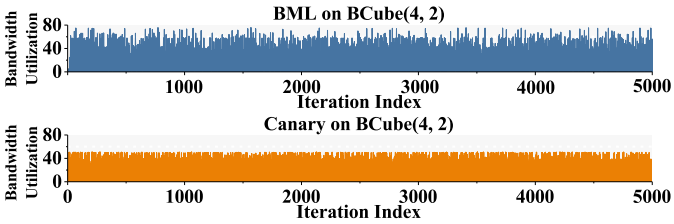
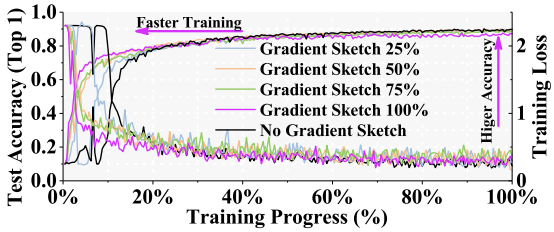


Fig. 21. The real-time bandwidth utilization of Canary on BCube(4, 2) is much less than that of BML-PT.

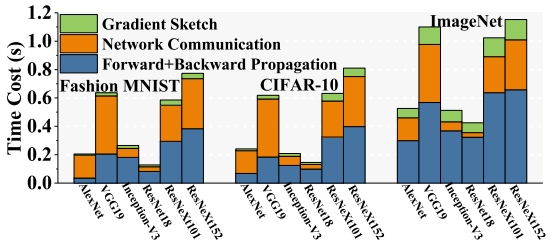
cluster scale in Fig. 20. Theoretically, the speedup of image processing speed follows a linear relationship with increase of tworker number, we use the black solid line to represent the linear scaling. Although the speedup of Canary does not strictly follow this linear relationship, Canary still provides a stable improvement under different workload configuration, especially when using more complex datasets. We can observe that the speedup of ImageNet is higher than that of other two datasets, because the computation of forward and backward propagation will dominate the per-iteration time in this scenario. Besides, the speedup of communication-intensive models with more parameters is usually lower than that of computation-intensive models. For example, Inception-V3 owns a better speedup than VGG19, mainly due to two reasons. First, the parameter size of Inception-V3 model (about 83.18 MB) is much smaller than that of VGG19 model (about 532.61 MB), making Inception-V3 yield less communication overhead. Second, Inception-V3 is more computation-intensive than VGG19, due to the large amount of CONV and pooling operations. Consequently, models with more computation demand while less communication overhead can achieve better speedup.

6.4 Communication Overhead

Measuring the communication overhead is significant to inspect the online performance of Canary . We will discuss how to measure the network traffic first. Recall that all the nodes are based on independent virtual machines, each of which owns different ENIs and private IP addresses. In our program codes of distributed DL training, each node sends and receives the gradient pieces via its ENIs. Therefore, we can measure the communication overhead of a node by checking the incoming and outgoing traffic of the ENIs. We use the Linux console application called `nload` [62] to monitor network traffic and bandwidth usage in real time. As to performance comparison of network communication, we mainly consider the real-time bandwidth utilization of our system and BML-PT in a 16-node BCube(4, 2) network on VGG19 model with CIFAR-10 dataset. As shown in Fig. 21, we can observe that Canary occupies less bandwidth utilization within the entire 5,000 seconds of network records, where the peak bandwidth utilization of Canary and BML-PT is 51.05 and 75.98 percent, respectively. Moreover, Canary yields less traffic burst while BML-PT's curve fluctuates more dramatically. This indicates that the asynchronous gradient exchange can provide better traffic offloading over BML's BSP. Actually, the accumulated padding area represents the total traffic amounts during DL training. In this aspect, Canary reduces the network traffic by up to 56.28 percent over BML-PT on average.



(a) The gradient sketch method accelerates the early-stage training speed while not degrading the final test accuracy.



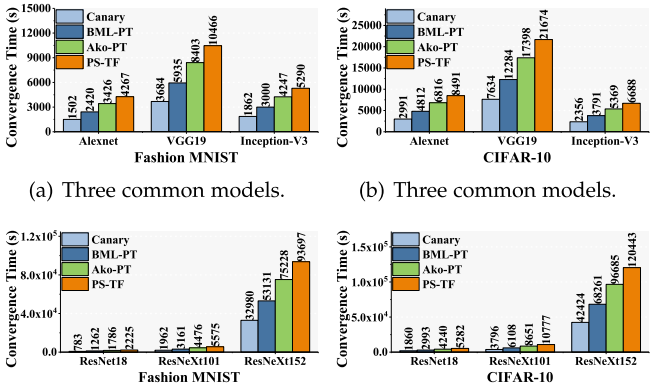
(b) The computational overhead (time cost in one iteration) of gradient sketch compared with other operations, using the GPU cluster of group (1).

Fig. 22. The gradient sketch method performs well in terms of training quality and computational cost.

6.5 Gradient Sketch Performance

As we reduce the traffic volume and communication overhead by compressing the gradient tensors in low-precision data type, it is necessary to check the training quality with gradient sketch. Therefore, we control the percentile of enabling gradient sketch during the training progress and compare the convergence curves under different configurations. The training is based on the AlexNet model with Fashion MNIST dataset. As shown in Fig. 22a, the percentile is adjusted from 0 percent (no gradient sketch) to 100 percent (always enable gradient sketch). We can observe that the top one test accuracy until convergence does not decrease too much. Meanwhile, the early-stage training speed is significantly accelerated by enabling gradient sketch (the pink line), compared with the original training configuration (the black line). This comparison shows that our gradient sketch method can effectively accelerate the convergence speed while not degrading the training quality.

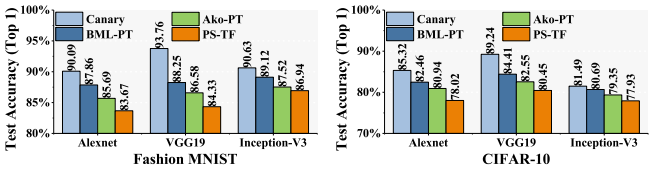
Besides, we analyse the computational overhead of gradient sketch by measuring the time cost of the three-step workflow, under different training workload. As shown in Fig. 22b, the time cost of operating gradient sketch (the green column) is relatively short, even on the deep ResNeXt152 model with ImageNet dataset, where the major overhead is from the operations of forward and backward propagation, and the network transmission of gradient tensors. This phenomenon indicates that the gradient sketch method will not bring much extra computational overhead to the distributed training system. Note that the efficiency of gradient sketch is impacted by the property of models, datasets and computing hardware. Therefore, we believe the gradient sketch method could be further optimized. For example, we could employ gradient sketch to certain layers instead of the whole model. Also, this method can be implemented via dedicated hardware, e.g., FPGA, to fully exploit the acceleration power. These are interesting directions for future work.



(a) Three common models. (b) Three common models.

(c) ResNet-based models. (d) ResNet-based models.

Fig. 23. Canary reduces convergence time on different workload using the 9-node cluster of group (2).



(a) Given time: 700, 1800 and 900 seconds for AlexNet, VGG-19 and Inception-V3, respectively.

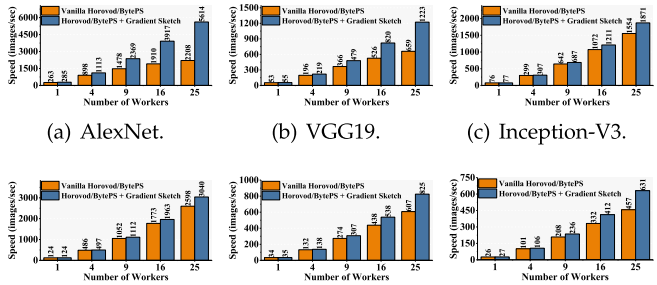
(b) Given time: 2000, 6000 and 1700 seconds for AlexNet, VGG-19 and Inception-V3, respectively.

Fig. 24. Baseline comparison of test accuracy after given training time using the 9-node cluster of group (2).

6.6 Baseline Comparison and System Improvement

As to the statistical efficiency of realistic deployment, we compare Canary with BML-PT and PS-TF in a 9-node cluster using group (2) GPUs. Note that Canary and BML-PT are deployed on a BCube(3, 2) network while PS-TF is deployed in a tree-based topology with 1 server and 8 workers. We consider two metrics: (1) training convergence time and (2) top one test accuracy after given training time. As shown in Fig. 23, Canary yields shorter convergence time on six models with two datasets. Specifically, under DL training configuration of $s = 18$ and $p = 6$ (total colayer number is 10) on VGG19 model with Fashion MNIST dataset, Canary is faster than BML-PT, Ako-PT and PS-TF by up to $1.61\times$, $2.28\times$ and $2.84\times$, respectively. Moreover, Canary achieves the highest test accuracy after the same given time in Fig. 24. For example, given 15 minutes on Inception-V3 model with Fashion MNIST dataset, Canary holds 90.63 percent top one test accuracy, which is higher than BML-PT and PS-TF.

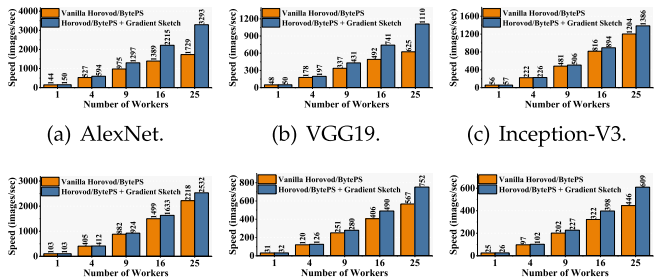
We also apply the idea of gradient sketch to the state-of-the-art frameworks (Horovod and BytePS) to check the system improvement of training efficiency. The evaluation is conducted on six models with three datasets, in BCube(4,2) network with 10 Gbps Ethernet. As shown in Figs. 25, 26 and 27, we focus on the image processing speed in one iteration under different cluster scale. As the ByteScheduler inside BytePS shares the same features from Horovod, these two systems own the similar performance. We observe that both Horovod and BytePS can improve their image processing speed by employing the gradient sketch method, especially when training the communication-intensive models with large parameter volume (e.g., VGG19 and ResNeXt152). The



(a) AlexNet. (b) VGG19. (c) Inception-V3.

(d) ResNet18. (e) ResNeXt101. (f) ResNeXt152.

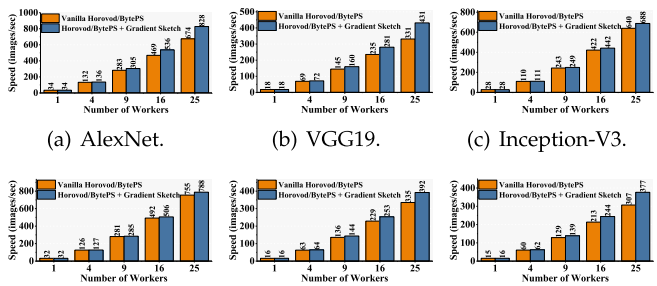
Fig. 25. The comparison of training Fashion MNIST dataset.



(a) AlexNet. (b) VGG19. (c) Inception-V3.

(d) ResNet18. (e) ResNeXt101. (f) ResNeXt152.

Fig. 26. The comparison of training CIFAR-10 dataset.



(a) AlexNet. (b) VGG19. (c) Inception-V3.

(d) ResNet18. (e) ResNeXt101. (f) ResNeXt152.

Fig. 27. The comparison of training ImageNet dataset.

key to achieve this improvement is the gradient sketch method can effectively compress gradient tensors by restricting the original Float32 data within 8 bits, so as to reduce the traffic size when exchanging the gradients. Consequently, the gradient sketch method inside Canary can apply to other distributed DL frameworks to accelerate the training speed.

7 Conclusion

We present Canary, a decentralized DL architecture conducting piece-level gradient exchange in multi-interface network. We conduct a co-design of traffic compression and parameter synchronization to fully exploit the capacity of multi-interface network without aggravating network bottleneck. The experimental results demonstrate that Canary outperforms commodity PS on TensorFlow, Ako on PyTorch and BML on PyTorch in terms of training convergence, scalability and communication overhead. Specifically, Canary reduces traffic volume, by up to 56.28 percent, on average, less than the BML on PyTorch. Besides, Canary accelerate training convergence, by up to $1.61\times$, $2.28\times$ and $2.84\times$ faster than BML on PyTorch, Ako on PyTorch and PS on

TensorFlow, respectively. Moreover, the gradient sketch method inside Canary can apply to existing learning frameworks and further improve the efficiency of distributed training.

Acknowledgment

This work was supported by the funding from Hong Kong RGC Research Impact Fund (RIF) with the Project No. R5060-19 and R5034-18, General Research Fund (GRF) with the Project No. 152221/19E, Collaborative Research Fund (CRF) with the Project No. C5026-18G, the National Natural Science Foundation of China (Grant No. 61872310, 61772286, 61802208, and 61872195), Jiangsu Key Research and Development Program (BE2019742), Natural Science Foundation of Jiangsu Province of China (BK20191381), and Natural Science Foundation for Distinguished Young Scholar of Jiangsu under Grant No. BK2020010062.

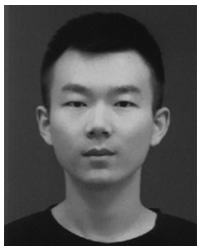
References

[1] J. Jiang, B. Cui, C. Zhang, and L. Yu, "Heterogeneity-aware distributed parameter servers," in *Proc. ACM Int. Conf. Manage. Data*, 2017, pp. 463–478.
[2] H. Zhang *et al.*, "Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2017, pp. 181–193.
[3] M. Li *et al.*, "Scaling distributed machine learning with the parameter server," in *Proc. 11th USENIX Conf. Operating Syst. Des. Implementation*, 2014, pp. 583–598.
[4] W. Xiao *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *Proc. 13th USENIX Conf. Operating Syst. Des. Implementation*, 2018, pp. 595–610.
[5] J. Chen, R. Monga, S. Bengio, and R. Józefowicz, "Revisiting distributed synchronous SGD," vol. abs/1604.00981, 2016, *arXiv*.
[6] S. Venkataramani *et al.*, "ScaleDeep: A scalable compute architecture for learning and evaluating deep networks," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, 2017, pp. 13–26.
[7] P. Moritz *et al.*, "Ray: A distributed framework for emerging AI applications," in *Proc. OSDI*, 2018, pp. 561–577.
[8] A. Harlap *et al.*, "PipeDream: Fast and efficient pipeline parallel DNN training," vol. abs/1806.03377, 2018, *arXiv*.
[9] Y. Huang *et al.*, "FlexPS: Flexible parallelism control in parameter server architecture," *Proc. VLDB Endowment*, vol. 11, pp. 566–579, 2018.
[10] E. P. Xing *et al.*, "Petuum: A new platform for distributed machine learning on big data," in *Proc. 21th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2015, pp. 1335–1344.
[11] T. Chen *et al.*, "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," vol. abs/1512.01274, 2015, *arXiv*.
[12] P. Watcharapichat, V. L. Morales, R. C. Fernandez, and P. R. Pietzuch, "Ako: Decentralised deep learning with partial gradient exchange," in *Proc. 7th ACM Symp. Cloud Comput.*, 2016, pp. 84–97.
[13] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. ACM SIGCOMM Conf. Data Commun.*, 2008, pp. 63–74.
[14] S. Wang *et al.*, "BML: A high-performance, low-cost gradient synchronization algorithm for DML training," in *Proc. 32nd Int. Conf. Neural Inf. Process. Syst.*, 2018, pp. 4243–4253.
[15] 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 *Proc. IEEE Conf. Comput. Commun.*, 2019, pp. 1729–1737.
[16] X. Lian, C. Zhang, H. Zhang, C. Hsieh, W. Zhang, and J. Liu, "Can decentralized algorithms outperform centralized algorithms? A case study for decentralized parallel stochastic gradient descent," in *Proc. 31st Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 5336–5346.
[17] C. Guo *et al.*, "BCube: A high performance, server-centric network architecture for modular data centers," in *Proc. ACM SIGCOMM Conf. Data Commun.*, 2009, pp. 63–74.
[18] G. Malewicz *et al.*, "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146.
[19] W. Zhang, S. Gupta, X. Lian, and J. Liu, "Staleness-aware async-SGD for distributed deep learning," in *Proc. 25th Int. Joint Conf. Artif. Intell.*, 2016, pp. 2350–2356.
[20] H. Cui *et al.*, "Exploiting bounded staleness to speed up big data analytics," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, 2014, pp. 37–48.
[21] Q. Ho *et al.*, "More effective distributed ML via a stale synchronous parallel parameter server," in *Proc. 26th Int. Conf. Neural Inf. Process. Syst.*, 2013, pp. 1223–1231.
[22] C. Chen, Q. Weng, W. Wang, B. Li, and B. Li, "Fast distributed deep learning via worker-adaptive batch sizing," in *Proc. ACM Symp. Cloud Comput.*, 2018, Art. no. 521.
[23] A. Harlap *et al.* "Addressing the straggler problem for iterative convergent parallel ML," in *Proc. 7th ACM Symp. Cloud Comput.*, 2016, pp. 98–111.
[24] H. Li, A. Kadav, E. Kruus, and C. Ungureanu, "MALT: Distributed data-parallelism for existing ML applications," in *Proc. 10th Eur. Conf. Comput. Syst.*, 2015, pp. 1–16.
[25] 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.
[26] Alibaba cloud, 2020. [Online]. Available: https://www.alibabacloud.com/
[27] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. 25th Int. Conf. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.
[28] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," vol. abs/1409.1556, 2014, *arXiv*.
[29] 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.*, 2016, pp. 2818–2826.
[30] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.
[31] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 5987–5995.
[32] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms," vol. abs/1708.07747, 2017, *arXiv*.
[33] A. Krizhevsky, "Learning multiple layers of features from tiny images," Univ. Toronto, Tech. Rep., 2012.
[34] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2009, pp. 248–255.
[35] A. Paszke *et al.*, "Automatic differentiation in PyTorch," in *Proc. NIPS Autodiff Workshop*, 2017, pp. 1–4.
[36] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proc. 12th USENIX Conf. Operating Syst. Des. Implementation*, 2016, pp. 265–283.
[37] A. Sergeev and M. D. Balso, "Horovod: Fast and easy distributed deep learning in tensorflow," vol. abs/1802.05799, 2018, *arXiv*.
[38] Baidu ring-allreduce: Bringing HPC techniques to deep learning, 2016. [Online]. Available: https://github.com/baidu-research/baidu-allreduce
[39] MPI forum: Message passing interface (MPI) forum home page, 2020. [Online]. Available: https://www.mpi-forum.org/
[40] Open MPI: Open source high performance computing, 2020. [Online]. Available: https://www.open-mpi.org/
[41] NVIDIA collective communications library, 2020. [Online]. Available: https://developer.nvidia.com/nccl
[42] Pytorch, 2020. [Online]. Available: https://pytorch.org
[43] J. Dean *et al.*, "Large scale distributed deep networks," in *Proc. 25th Int. Conf. Neural Inf. Process. Syst.*, 2012, pp. 1223–1231.
[44] F. Niu, B. Recht, C. Re, and S. J. Wright, "HOGWILD!: A lock-free approach to parallelizing stochastic gradient descent," in *Proc. 24th Int. Conf. Neural Inf. Process. Syst.*, 2011, pp. 693–701.
[45] N. Mellempudi, A. Kundu, D. Das, D. Mudigere, and B. Kaul, "Mixed low-precision deep learning inference using dynamic fixed point," vol. abs/1701.08978, 2017, *arXiv*.
[46] Z. Cai, X. He, J. Sun, and N. Vasconcelos, "Deep learning with low precision by half-wave Gaussian quantization," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 5406–5414.

[47] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. 2nd USENIX Conf. Hot Topics Cloud Comput.*, 2010, Art. no. 10.
[48] X. Meng *et al.*, "MLlib: Machine learning in apache spark," *J. Mach. Learn. Res.*, vol. 17, pp. 34:1–34:7, 2016.
[49] T. M. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project Adam: Building an efficient and scalable deep learning training system," in *Proc. 11th USENIX Conf. Operating Syst. Des. Implementation*, 2014, pp. 571–582.
[50] P. MacArthur, Q. Liu, R. D. Russell, F. Mizero, M. Veeraraghavan, and J. M. Dennis, "An integrated tutorial on InfiniBand, verbs, and MPI," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 4, pp. 2894–2926, Fourth Quarter 2017.
[51] H. Zhang, K. Kara, J. Li, D. Alistarh, J. Liu, and C. Zhang, "ZipML: An end-to-end bitwise framework for dense generalized linear models," vol. abs/1611.05402, 2016, *arXiv*.
[52] J. Jiang, F. Fu, T. Yang, and B. Cui, "SketchML: Accelerating distributed machine learning with data sketches," in *Proc. ACM Int. Conf. Manage. Data*, 2018, pp. 1269–1284.
[53] M. Greenwald and S. Khanna, "Space-efficient online computation of quantile summaries," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2001, pp. 58–66.
[54] Y. Peng *et al.*, "A generic communication scheduler for distributed DNN training acceleration," in *Proc. 27th ACM Symp. Operating Syst. Princ.*, 2019, pp. 16–29.
[55] C. Louizos, M. Reisser, T. Blankevoort, E. Gavves, and M. Welling, "Relaxed quantization for discretized neural networks," vol. abs/1810.01875, 2018, *arXiv*.
[56] D. Anderson, P. Bevan, K. J. Lang, E. Liberty, L. Rhodes, and J. Thaler, "A high-performance algorithm for identifying frequent items in data streams," in *Proc. Internet Meas. Conf.*, 2017, pp. 268–282.
[57] L. D. Lathauwer, B. D. Moor, and J. Vandewalle, "A multilinear singular value decomposition," *SIAM J. Matrix Anal. Appl.*, vol. 21, pp. 1253–1278, 2000.
[58] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *ACM Queue*, vol. 6, no. 2, pp. 40–53, 2008.
[59] PyTorch distributed communication package, 2020. [Online]. Available: https://pytorch.org/docs/stable/distributed.html
[60] Alibaba cloud elastic GPU service, 2020. [Online]. Available: https://www.alibabacloud.com/product/gpu
[61] Alibaba cloud elastic network interface, 2020. [Online]. Available: https://www.alibabacloud.com/help/doc-detail/58496.htm
[62] nload: Displays the current network usage, 2012. [Online]. Available: https://linux.die.net/man/1/nload

Qihua Zhou (Student Member, IEEE) received the BS degree in information security from the Nanjing University of Posts and Telecommunications, Nanjing, China, in 2015. His research interests include distributed systems, parallel computing, on-device learning, and domain-specific hardware acceleration.

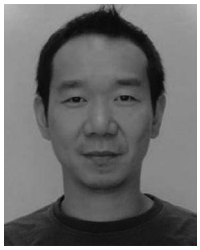
Kun Wang (Senior Member, IEEE) received the two PhD degrees in computer science from the Nanjing University of Posts and Telecommunications, Nanjing, China, in 2009, and from the University of Aizu, Aizuwakamatsu, Japan, in 2018. He was a post-doctoral fellow with the UCLA, Los Angeles, California from 2013 to 2015, where he is a senior research professor now. He was a research fellow with the Hong Kong Polytechnic University, Hong Kong, from 2017 to 2018, and a professor with the Nanjing University of Posts and Telecommunications. His current research interests are mainly in the area of big data, wireless communications and networking, energy Internet, and information security technologies.



Haodong Lu (Student Member, IEEE) is currently working toward the MSc degree at the School of Internet of Things, Nanjing University of Posts and Telecommunications, Nanjing, China. His current research interests include distributed processing, parallel computing, and deep learning.

Wenyao Xu (Senior Member, IEEE) received the bachelor's and master's degrees in computer science from Zhejiang University, Hangzhou, China, and the PhD degree in electronic engineering from the University of California, Los Angeles, Los Angeles, California. He is an associate professor with tenure at the Computer Science and Engineering Department, State University of New York at Buffalo, Buffalo, New York. His group has focused on exploring novel sensing and computing technologies to build up innovative Internet-of-Things (IoT) systems for highimpact human-technology applications in the fields of Smart Health, and Cyber-Security. Results have been published in peer-reviewed top research venues across multiple disciplines, including computer science conferences (e.g., ACM MobiCom, SenSys, MobiSys, UbiComp, ASPLOS, ISCA, HPCA, and CCS), biomedical engineering journals (e.g., the *IEEE Transactions on Biomedical Engineering*, *IEEE Transactions on Biomedical Circuits and Systems*, and *IEEE Journal of Biomedical and Health Informatics*), and medicine journals (e.g., the *Lancet*). To date, his group has published more than 160 papers, won six best paper awards, two best paper nominations, and three international best design awards. His inventions have been filed within U.S. and internationally as patents, and have been licensed to industrial players. His research has been reported in high-impact media outlets, including the Discovery Channel, CNN, NPR, and the Wall Street Journal. He is serving as an associate editor of the *IEEE Transactions on Biomedical Circuits and Systems*, and on technical program committees of numerous conferences in the field of Smart Health and Internet of Things, and was a TPC co-chair of the *IEEE Body Sensor Networks*, in 2018.

Yanfei Sun (Member, IEEE) is a full professor with the School of Automation and School of Artificial Intelligence, Nanjing University of Posts and Telecommunications, Nanjing, China. He is also a director of the Jiangsu Engineering Research Center of HPC and Intelligent Processing, Nanjing, China. His current research interests are mainly in the area of future networks, Industrial Internet, energy Internet, big data management and analysis, and intelligent optimization and control.



Song Guo (Fellow, IEEE) is a full professor and associate head with the Department of Computing, Hong Kong Polytechnic University. His research interests are mainly in the areas of big data, cloud computing, mobile computing, and distributed systems. He is the recipient of the 2019 IEEE TCBD Best Conference Paper Award, 2018 IEEE TCGCC Best Magazine Paper Award, 2017 IEEE Systems Journal Annual Best Paper Award, and other eight best paper awards from IEEE/ACM conferences. He was an IEEE ComSoc distinguished lecturer and served in IEEE ComSoc Board of Governors. He has also served as general and program chair for numerous IEEE conferences. He is the editor-in-chief of the *IEEE Open Journal of the Computer Society*, and an associate editor of the *IEEE Transactions on Cloud Computing*, *IEEE Transactions on Sustainable Computing*, and *IEEE Transactions on Green Communications and Networking*.