Accelerating Distributed DNN Training via Transport Layer Scheduling

Qingyang Duan, Chao Peng, Zeqin Wang, Yuedong Xu, Shaoteng Liu, Jun Wu, *Senior Member, IEEE*, and John C. S. Lui, *Fellow, IEEE*

***Abstract*—Communication scheduling is crucial to accelerate the training of large deep learning models, in which the transmission order of layer-wise deep neural network (DNN) tensors is determined for a better computation-communication overlap. Prior approaches adopt user-level tensor partitioning to enhance the priority scheduling with finer granularity. However, a startup time slot inserted before every tensor partition will neutralize this scheduling gain. Tuning hyper-parameters for tensor partitioning is difficult, especially when the network bandwidth is shared or time-varying in multi-tenant clusters. In this article, we propose Mercury, a simple transport layer scheduler that moves the priority scheduling to the transport layer at the packet granularity. The packets with the highest priority in the Mercury buffer will be transmitted first. Mercury achieves the near-optimal overlapping between communication and computation. It also leverages the immediate aggregation at the transport layer to enable the full overlapping of gradient push and pull. We implement Mercury in MXNet and conduct comprehensive experiments on five popular DNN models in various environments. Mercury can well adapt to dynamic communication and computation resources. Experiments show that Mercury accelerates the training by up to 130% compared to the classical PS architecture, and 104% compared to state-of-the-art tensor partitioning methods.**

***Index Terms*—Computation-communication overlap, distributed machine learning, parameter server, transport layer scheduling.**

I. Introduction

THE past decade has witnessed the tremendous success of Deep Neural Networks (DNNs) in various applications, such as natural language processing [1], [2], computer vision [3], speech recognition [4], recommendation [5], [6] etc. With the growth of data volume and model size, distributed deep learning is becoming the norm of modern machine learning systems. To run various training tasks efficiently, large companies are building shared multi-tenant GPU clusters with many GPU and CPU machines. Parameter Server (PS) architecture [7] with data parallelism is widely used in distributed DNN training where data samples are distributed among a number of workers. Each worker performs the local computation iteratively, while synchronizes the results with the PSes at the end of every iteration. However, network bandwidth is usually the bottleneck of distributed DNN training [8], [9]. The transmission of model gradients or parameters consumes a large proportion of training time. Therefore, improving the communication efficiency is essential to accelerate distributed DNN training.

DNN training procedure is generally characterized as a Directed Acyclic Graph (DAG) in existing machine learning frameworks, such as MXNet [10], PyTorch [11] and TensorFlow [12]. Each iteration at a worker consists of two layer-wise computation operations (forward and backward propagation) and two sequential communication operations (push and pull) in which their interdependency is determined by the underlying DAG. They consume different types of resources so that the training time will be reduced if communication can overlap with computation, and push can overlap with pull. The approach of scheduling communication operations has been extensively studied. Wait-free backpropagation (WFBP) [13], adopted by prevalent DNN frameworks (e.g., MXNet), transmits the gradient tensor of a layer after the completion of its backward propagation, resulting in a relatively low degree of such overlapping. TicTac [14] derives the optimal transmission order of layer-wise tensors through critical-path analysis on the underlying DAG.

Recently, tensor partitioning has been proposed to improve the granularity of priority scheduling and pipelining, which delivers a much better computation-communication overlap than the baseline with intact tensors. P3 [15] segments DNN layers into smaller slices and synchronizes them based on their priorities. ByteScheduler [16], as a generic communication scheduler, is framework agnostic and communication method-agnostic. It adopts a Bayesian optimization approach to tune the partition size for adapting to various training models and system configurations. AutoByte [17] uses a meta-network learning approach to search effective hyper-parameters at runtime for ByteScheduler. BytePS [18] presents a unified communication architecture, leveraging spare CPU and bandwidth resources in the cluster to accelerate the DNN training. A multi-stage pipelining method

Manuscript received 19 July 2022; revised 30 January 2023; accepted 14 February 2023. Date of publication 28 February 2023; date of current version 4 April 2023. This work was supported in part by the Natural Science Foundation of China under Grant 62072117, in part by the Key-Area Research and Development Program of Guangdong Province under Grant 2020B010166003, in part by the Shanghai Natural Science Foundation under Grant 22ZR1407000. Recommended for acceptance by S. Wang. *(Corresponding author: Yuedong Xu.)*

Qingyang Duan, Chao Peng, Zeqin Wang, and Yuedong Xu are with the Department of Electronic Engineering, Fudan University, Shanghai 200437, China (e-mail: duanqy20@fudan.edu.cn; pchao321@163.com; wangzeqin17@fudan.edu.cn; ydxu@fudan.edu.cn).

Shaoteng Liu is with the 2012 Lab, Huawei Technologies Co. Ltd., Shenzhen, Guangdong 518129, China (e-mail: liushaoteng@huawei.com).

Jun Wu is with the School of Computer Science and Technology, Fudan University, Shanghai 200437, China (e-mail: wujun@fudan.edu.cn).

John C. S. Lui is with the Computer Science & Engineering, The Chinese University of Hong Kong,, Hong Kong (e-mail: cslui@cse.cuhk.edu.hk).

Digital Object Identifier 10.1109/TPDS.2023.3250462

is deployed with tensor partitioning that overlaps the processing time of each optimization step. P3, ByteScheduler, AutoByte and BytePS are all implemented ***above*** the message-based communication library, which we refer to as user-level tensor partitioning methods.

Though performant, these user-level tensor partitioning methods can hardly achieve the near-optimal computation-communication overlap in the presence of the nontrivial communication stack-induced overhead. This overhead, termed as *startup time*, is inserted before the push operation of every tensor partition. The priority scheduling is more fine-grained if a tensor is partitioned into a number of slices. However, more slices bring higher total startup time, yielding poor bandwidth utilization. The performance gain achieved by efficient priority scheduling is neutralized accordingly. Meanwhile, the user-level tensor partitioning methods can hardly adapt to dynamically changing resources because the careful tuning of hyper-parameters is required to tackle different runtime environments. ByteScheduler develops a Bayesian optimization approach to search the optimal hyper-parameters (i.e., partition size and credit size), yet it is not suitable in dynamic environments due to its minute-level search time. More essentially, this dilemma is inherent in all user-level tensor partitioning methods.

In this paper, we propose Mercury (meaning "fast"), a transport layer scheduler to achieve near-optimal scheduling without additional communication overhead. Mercury possesses three merits. First, the system architecture of Mercury is ***simple***. Mercury comprises two major components, the packet-level priority scheduler at the worker and the packet-level aggregator at the PS. The priority queue is realized using multiple FIFO queues in the transport layer where each queue corresponds to a certain priority. The tensor packets with the highest priority are transmitted first. The transport layer of the PS is in charge of aggregating gradients in Mercury packets in a pipelining manner. Second, Mercury is highly ***efficient***. No tensor partitioning is allowed in Mercury so that there is no throughput penalty caused by the startup time. The priority scheduling of tensors that facilitates the overlapping of computation and communication is realized at nearly minimal granularity. Meanwhile, the aggregation at the packet level enables the nearly perfect overlapping of push and pull. Third, Mercury provides a ***generic*** transport layer distributed training acceleration method. Only the point-to-point communication module of DNN frameworks is modified, and other user-level acceleration techniques (e.g., tensor fusion, compression, etc.) can be incorporated as well.

We evaluate Mercury with extensive experiments under various runtime environments. Two recent user-level tensor partitioning methods, ByteScheduler and BytePS, serve as our baselines. Experimental results across representative DNN models demonstrate that in a static environment, Mercury outperforms native PS architecture and user-level tensor partitioning methods by up to 130% and 74%, respectively. Mercury can adapt well to the dynamic bandwidth and the dynamic computing power, i.e. outperforming native PS architecture by up to 129% and user-level tensor partitioning methods by up to 104% in terms of the training speed.

The rest of this paper is organized as follows. Section II introduces the necessary background of distributed training and communication scheduling. Section III reveals the weaknesses of existing tensor partitioning methods. Section IV presents the system architecture and the detailed design. Section V provides the trace-driven analysis of the optimal training speedup. Section VI evaluates the performance of Mercury in our testbed. Section VII describes the related work, and Section VIII concludes this paper.

II. Background

In this section, we introduce the background of distributed DNN training, in particular, the workflow of distributed training, the communication bottleneck and the commonly-used communication scheduling strategies.

A. Distributed DNN Training

Data Parallelism. The training of DNN models on a single server is usually time inefficient with the growth of data volume. A wise approach is to partition the data into multiple shards, each of which is placed and computed at an individual worker. This is called distributed training with data parallelism. All workers iteratively train the local model and a minibatch of data samples is used by a worker in each iteration. More precisely, the workers conduct forward propagation (FP) to calculate the loss function, and then conduct backward propagation (BP) to compute the gradients. Bulk Synchronous Parallel (BSP) [19] scheme is widely used to coordinate their collaborative training. They synchronize the local gradients (or parameters) at the end of each iteration to generate the global up-to-date model as the starting point of the next iteration. Parameter server and all-reduce are two important communication architectures where the former is more frequently adopted.

Parameter Server Architecture. Some machines in the PS architecture are logically categorized as "parameter servers (PSes)" that maintain the global consensus of model parameters. Each training iteration operates as follows: a) each worker computes the local gradients (and averages gradients of local GPUs if this worker contains multiple GPUs); b) each worker "pushes" the gradients to the PSes; c) the PSes aggregate the gradients and generate the global model parameters; d) each worker "pulls" the global model parameters from the PSes (and broadcasts them to local GPUs if this worker contains multiple GPUs). The communication takes place only between the PSes and the workers while the workers themselves do not communicate with each other.

PS architecture has better performance than all-reduce because it can utilize extra CPU machines as PSes. Supposing there are n GPU machines and n CPU machines and the model size is S. In one iteration of PS architecture, each GPU (and CPU) machine sends (and receives) S traffic to the network. While using all-reduce, each GPU machine sends (and receives) $2(n-1)S/n$ unit of traffic [18]. Thus PS architecture can be more time-efficient.

Bottleneck of Distributed Training. The recent trend of distributed machine learning is to employ advanced hardware to

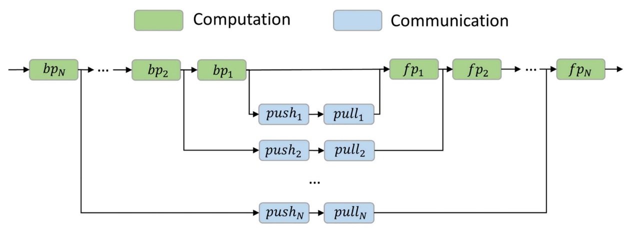


Fig. 1. Dependency DAG of distributed DNN training.

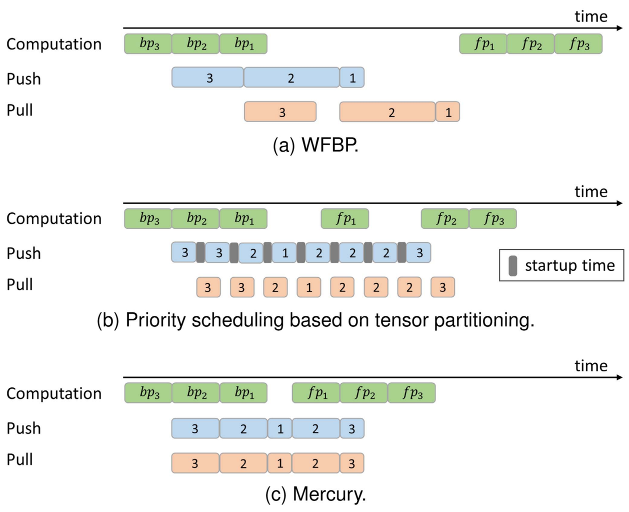


Fig. 2. An example showing one iteration time under different scheduling strategies.

compute large DNN models, while the upgrading of network bandwidth is rather slow. Hence, the ratio of communication time to computation time becomes very high, throttling the efficiency of distributed training. For instance, VGG-16 contains about 135 million parameters at the size of 540 MB so that a round of push-pull communication needs 0.864 s in a 10 Gbps network, but a local iteration only takes 0.580 s in a NVIDIA GTX 1080Ti GPU. This implies that the communication time overwhelms the local computing time, encumbering the efficiency of model training. Improving the network hardware can relieve the problem of communication bottleneck. Considering that the peak performance of new GPUs is increasing rapidly in the past few years, we believe that reducing the model transmission time is a crucial challenge in the heart of distributed DNN training.

B. Communication Scheduling

Computation-Communication Dependency Graph. The computation and communication operations of DNNs can be modeled as a DAG. Fig. 1 illustrates the layer-wise dependency of operations in two consecutive iterations. Let $fp_i, bp_i, push_i$, and $pull_i$ be the FP, BP, push and pull of layer i. FPs are executed from the first layer to the last layer while BPs are executed in an opposite direction. Hence, fp_i relies on fp_{i-1} and $pull_i$, $pull_i$ relies on $push_i$, $push_i$ relies on bp_i, and bp_i relies on bp_{i+1} [16]. Without loss of generality, we can refer to all the operations in Fig. 1 as one iteration. Note that bp and fp consume computing resources, while $push$ uses uplink bandwidth and $pull$ uses downlink bandwidth. Therefore, the training time can be effectively reduced if the computation time overlaps with the communication time, and the push time overlaps with the pull time.

Let us illustrate the time usage of one iteration in Fig. 2 using a three-layer DNN as an example. We make a few simplifications for a better narrative: i) the gradients or parameters of one layer are represented as one tensor; ii) the push (resp. pull) time of a tensor is identical across workers; iii) the aggregation of gradients in PSes accomplishes instantly. Denote by TS_i the i^{th} tensor for $i \in \{1, 2, 3\}$. Different tensors may vary in size. Without loss of generality, we suppose TS_2 is larger than TS_3, and TS_3 is larger than TS_1.

WFBP is the default approach adopted by prevalent deep learning frameworks (e.g., MXNet). The gradient tensor of a layer is scheduled for transmission once its BP operation completes. In Fig. 2(a), after bp_3 is finished, the worker pushes the TS_3 gradient tensor to the PSes. The worker subsequently pulls the TS_3 parameter tensor and pushes the TS_2 gradient tensor, and so on and so forth. One can easily observe that WFBP diminishes the opportunities of computation-communication overlapping.

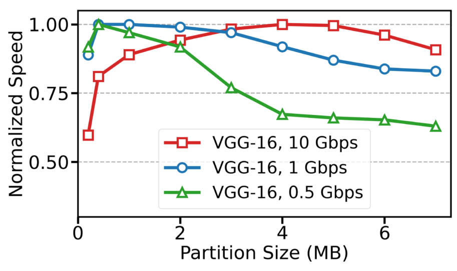
User-Level Tensor Partitioning Methods. From Fig. 1, we can observe that tensors with smaller layer numbers should be prioritized for transmission. A way to realize priority scheduling is via tensor partitioning. In distributed DNN training, each layer's tensor can be subdivided into multiple smaller partitions for fine-grained priority scheduling. In Fig. 2(b), the TS_2 and TS_3 tensors are partitioned into four and three smaller pieces, respectively. The tensor with a smaller index possesses a higher scheduling priority. After two rounds of $push_3$, the gradient tensors TS_2 are already available so that $push_2$ can be scheduled. When $push_2$ finishes, the gradient tensor TS_1 is ready so that the scheduler preempts to execute $push_1$ while pausing the remaining $push_2$ and $push_3$ operations. We will talk about the startup time later. One can see that the priority scheduling based on tensor partitioning can achieve better overlap between computation and communication and between push and pull.

III. Observations and Opportunities

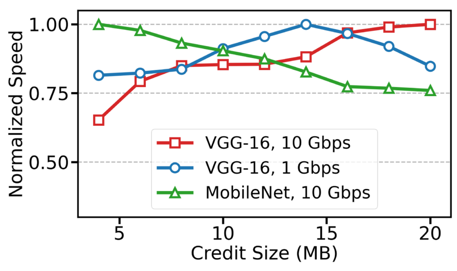
In this section, we show experimentally that the performance of user-level tensor partitioning methods largely depends on the careful tuning of hyper-parameters. We reveal their inefficiency and discuss new opportunities of reducing model synchronization time.

A. Inefficient Hyper-Parameter Tuning

In theory, the smallest partition size yields the best flexibility of priority scheduling. However, we can not neglect the communication overhead caused by tensor partitioning. Real-world systems usually take extra time between delivering two nearby



(a) Partition size vs normalized speed.



(b) Credit size vs normalized speed.

Fig. 3. Training VGG-16 and MobileNet in MXNet with ByteScheduler. The training speed is normalized by the highest value in each scenario. The credit size is tuned to be optimal in (a), and the partition size is tuned to be optimal in (b).

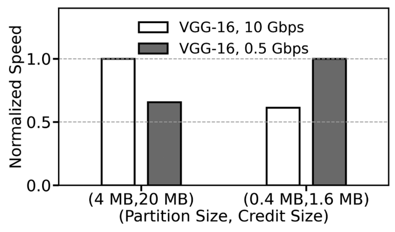


Fig. 4. Training VGG-16 in MXNet with ByteScheduler. (4 MB, 20 MB) is the optimal hyper-parameter combination for training VGG-16 in a 10 Gbps network and (0.4 MB, 1.6 MB) is that in a 0.5 Gbps network.

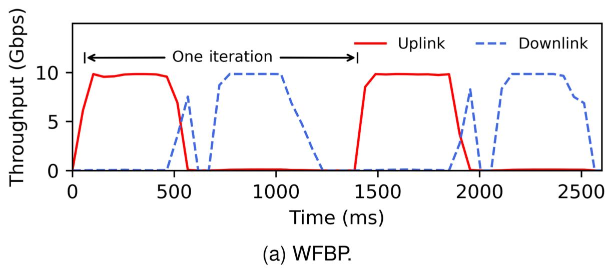
tensor partitions in a stop-and-wait manner. We refer to this extra time as *startup time*, as illustrated in Fig. 2(b). The startup time is irrelevant to the size of the tensor partition. Though extremely small (less than $1\tilde{}ms$ estimated in our testbed), we cannot neglect the accumulation of startup time with many tensor partitions. We will describe how the startup time is produced in detail in Section IV-A. Here, we only focus on the performance degradation caused by the startup time.

Partition Size Tuning. With a smaller partition size, each tensor is sliced into more partitions in each iteration. More startup slots are inserted in communication operations, which leads to heavier communication overhead. Therefore, it is necessary to configure a proper partition size beforehand to balance the scheduling efficiency and communication overhead. To evaluate the impact of the partition size and network bandwidth on the training speed, we train VGG-16 and MobileNet [20] with one worker and one PS interconnected by different bandwidth settings. We use ByteScheduler, a state-of-the-art user-level tensor partitioning method. In Fig. 3(a), one can observe that each bandwidth setting has a near “optimal” partition size, and the optimal sizes differ in different settings. If the partition size remains the same, the speed degrades severely when the bandwidth changes. For example, the normalized speed degrades by 9% (or 32%) when the bandwidth decreases from 10 Gbps to 1 Gbps (or 0.5 Gbps). Considering that the training is usually conducted in shared clusters, the partition size needs dynamic tuning to adapt to time-varying network bandwidth.

Credit Size Tuning. To realize preemption, tensor partitions are transmitted in a stop-and-wait manner. The worker sends one partition to the underlying FIFO transmission queue (e.g., the queue in ZeroMQ). The destination PS sends back a message notifying the completion of the partition’s transmission. After receiving this acknowledgement message, the worker schedules the next partition to the underlying transmission queue. In this way, the preemption granularity is the partition size. However, the startup time of the next partition cannot overlap with the transmission time of the current partition, causing link under-utilization. One possible approach to alleviate this problem is via credit-based preemption. The credit size works as a sliding window. Multiple tensor partitions are allowed to be delivered concurrently to the underlying FIFO transmission queue. These partitions are called unacknowledged partitions. The credit size is the maximum allowed total size of unacknowledged partitions. An unacknowledged partition is acknowledged after receiving its completion message. The worker can schedule new user-level partitions into the underlying transmission queue within the limitation of the credit size. If the credit size is set equal to the partition size, the credit-based preemption degenerates to the stop-and-wait preemption. Usually, the credit size is set as the multiple of the partition size.

The network utilization is improved because the transmission time of early tensor partitions overlaps with the startup time of subsequent partitions. However, the priority-based scheduling is undermined by a large credit for that the tensor partitions delivered to the underlying queue cannot be preempted anymore. In Fig. 3(b), we illustrate how the training speed is influenced by the credit size across different bandwidth settings and different DNN models. Different DNN models have different distributions of tensor size that affect the choice of optimal credit size. Similarly, adaptive tuning of credit size is indispensable. We further evaluate the impact of two hyper-parameters together in Fig. 4. The optimal pair of partition size and credit size may degrade the training speed by 40% when the network bandwidth changes.

ByteScheduler utilizes a Bayesian optimization approach to search for the optimal hyper-parameters (i.e., partition size and credit size) at the beginning of the training. These hyper-parameters remain fixed till the end of the training. This static configuration approach can be extended to auto-tuning by triggering the searching algorithm periodically. However, the search time of Bayesian optimization is non-negligible, e.g., hundreds of iterations (i.e., more than 10 minutes for training VGG-16 on a 10 Gbps link).



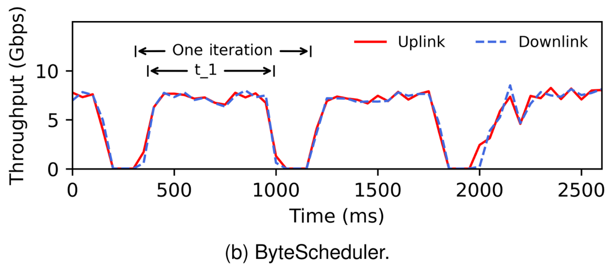


Fig. 5. Training VGG-16 with 2 workers and 2 PSes using MXNet. The network bandwidth is 10 Gbps.

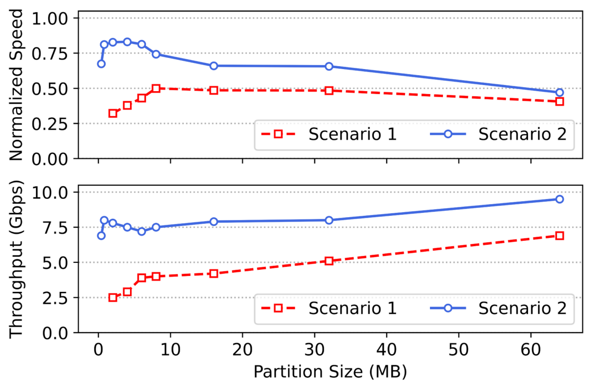


Fig. 6. Network throughput and training speed for ByteScheduler with different partition sizes: credit size equal to partition size (scenario 1) and optimal credit size (scenario 2).

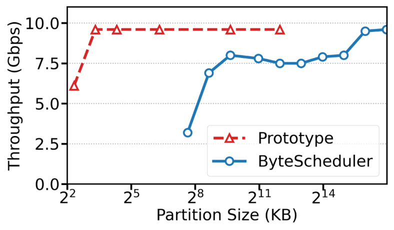


Fig. 7. Potential scheduling granularity in a 10 Gbps network.

B. *Insufficient Bandwidth Utilization*

We hereby show that even a well-tuned ByteScheduler suffers from inefficient bandwidth utilization in static network environments. Fig. 5(a) and (b) show the uplink and downlink throughput of a worker with WFBP and ByteScheduler, respectively. The throughput statistics are measured every 50 ms. WFBP is the default tensor scheduler of MXNet. In Fig. 5(b), the partition size and credit size for ByteScheduler are optimally tuned. Push and pull operations are executed during t_1. Compared with WFBP, ByteScheduler has a shorter duration per iteration, signifying its better training efficiency. However, the average network throughput during t_1 is only $\sim 73\%$ even with the "optimal" partition size and credit size. We can safely claim that the low bandwidth utilization is caused by the startup time.

To verify whether this problem can be solved by tuning hyper-parameters, we train the model under ByterScheduler with different hyper-parameter combinations in a 10 Gbps network with 2 workers and 2 PSes. Fig. 6 shows the trade-off between the network throughput and normalized training speed. The training speed is normalized by that of training with one GPU locally. The network throughput is averaged over the uplink traffic when push operations are executed. In order to analyze the impact of partitioning independently, we set the credit size equal to the partition size to bypass the credit-based preemption in scenario 1. In scenario 1, a larger partition size achieves higher bandwidth utilization. But large partition size implies large preemption granularity, leading to poor overlapping of computation and communication. So from the training speed's perspective, an "optimal" partition size is $\sim$ 10 MB. In scenario 2, for each experiment of a partition size, the credit size is tuned to maximize the training speed. From the training speed's perspective, an "optimal" combination of partition size and credit size is (4 MB, 20 MB). We can see that the credit-based preemption improves both bandwidth utilization and training speed. However, the network bandwidth is still underutilized when both scenarios reach the "optimal" point. This dilemma leaves a potential improvement room for more efficient scheduling strategies.

C. *Necessity of Transport Layer Scheduling*

In light of the above observations, we argue that these problems are inherent in all user-level tensor partitioning methods. A direct remedy is to develop better hyper-parameter tuning algorithms, yet leaving the startup overhead unresolved. An upper-layer solution can hardly achieve the optimal communication and computation overlapping without sacrificing the bandwidth utilization.

The underlying transport layer is capable of more fine-grained scheduling. To quantify the potential overhead of transport layer operations, we implement a simple partitioning prototype over TCP to send a big tensor with FIFO transmission scheduling. Fig. 7 shows the bandwidth utilization under the 10 Gbps network. Compared with ByteScheduler, this prototype can provide much smaller scheduling granularity (i.e., $\sim$ 100 times smaller than ByteSchduler) and fully utilize the bandwidth. Motivated by these findings, we design Mercury that provides fine-grained scheduling at the transport layer, and achieves near-optimal overlapping between computation and communication.

IV. Mercury Design

In this section, we first present key ideas of Mercury design. We then introduce the overview of system architecture and describe the implementation of Mercury.

A. Key Ideas

Inspired by our observations in Section III, we attempt to diagnose the root cause of startup time and come up with a set of key ideas of mitigating its adverse impacts.

The Root Cause of Startup Time. Distributed training systems adopt a similar communication stack. From top to bottom, the communication stack includes 1) *user code*, 2) *engine core*, 3) *message-based communication libraries*, and 4) *transport layer protocol* (e.g., TCP or RDMA). User code encapsulates computation, push and pull operations and their associated tensors into operators. Engine core determines the execution order of communication and computation operators according to their dependencies. It also schedules independent operators to be executed in parallel if possible. The message-based communication libraries usually consist of RPC, ZeroMQ, MPI, NVIDIA NCCL [21] and so on. They transmit tensors in a FIFO manner.

User-level tensor partitioning methods conduct slicing and scheduling in *user code*. More specifically, a push operator of a large tensor is partitioned into multiple operators (i.e., each is responsible for a tensor partition). These operators will be held in *user code* waiting for scheduling to the communication stack. Therefore, the startup time consists of 1) message-level acknowledgement delay of the last tensor partition, 2) scheduler delay (i.e., the time to choose the tensor partition with the highest priority), 3) engine core scheduling delay, and 4) message-based communication library pre-processing delay (e.g., RPC serialization, ZeroMQ queuing) chronologically. When the PS finishes receiving the data of a tensor partition, it sends an acknowledgement message to the worker. Then the worker knows that the transmission of this tensor partition finishes and schedules the next one or updates the credit window. This message-level acknowledgement delay is non-negligible. The scheduler delay is expected to be the smallest while the other three delays are notable for the startup time.

Key idea 1: Startup time minimization. In Mercury, a tensor (or a push operator) is delivered to the underlying transport layer directly as soon as the corresponding BP operation completes. Tensor partitioning does not exist at user code so that no startup time is inserted into push operations. Our proposal avoids the excessive startup time intervals caused by tensor partitioning at *user code*.

Key idea 2: Transport layer scheduling. Mercury implements the priority scheduling at the transport layer by configuring multiple transmission queues. The Mercury packets of the i^{th} tensor are stored at the i^{th} FIFO queue, and those with the highest priority are always transmitted first. Compared with user-level tensor partitioning methods, the priority scheduling and the preemption are put into effect at the packet granularity. The tensor of each layer has its unique priority by default, and multiple tensors can share the same priority when the number of layers is too large. Mercury's scheduling mechanism has to be carefully designed so that the overhead to choose the packet with the highest priority does not impair the throughput.

Key idea 3: Immediate PS aggregation. Mercury provides the immediate PS aggregation that guarantees the near-optimal overlapping of push and pull in the absence of tensor partitioning. The PS aggregates the gradients contained in a Mercury packet upon its arrival, and sends them back in a packet once the aggregation on all workers' gradients accomplishes. In another word, the pull operation is triggered at the packet granularity. The aggregation happens at the transport layer to avoid extra processing delay of the user-level communication stack. The computing logic of the aggregation has to be as simple as possible to reduce the processing time of each packet. Thus, we propose that the PS only sums up local gradients, and let optimizers (e.g., SGD or Adam) execute at the workers' GPUs. This speeds up the computation of optimizers and reduces the CPU load in PSes. This also makes Mercury a generic scheduler that supports almost all training algorithms.

Illustration of Potential Benefits. The time usage of our design, Mercury (meaning *fast*), is illustrated in Fig. 2(c). Preemption happens immediately after bp_2 and bp_1 finish. The push and pull operations are expected to be fully overlapped, which will be justified by experimental results in Section VI. There are no extra startup time slots in push operations. which means that the network bandwidth is fully utilized. The iteration time reduction in Mercury mainly comes from the more fine-grained priority scheduling and the higher bandwidth utilization.

B. Architecture Overview

Mercury has different designs in the worker and the PS. Fig. 8 shows the interactions between worker nodes and PS nodes during the training. *Worker Computing Engine* and *PS Computing Engine* are high-level training abstractions implemented by current deep learning frameworks. Worker Computing Engine performs BP and FP iteratively on GPUs and averages their gradient tensors (i.e., through local NVLink or PCIe). PS Computing Engine supports gradient or parameter synchronization. They deliver control messages by *Message-level Communication Library* to maintain connections and convey commands. *Mercury Worker* and *Mercury PS* only transmit gradient tensors between worker nodes and PS nodes. They are the two key components of Mercury's design. Traditional deep learning frameworks transmit all traffic using the message-level communication library. We distinguish the traffic into control messages and gradient tensors so that Mercury Worker and Mercury PS can perform different operations on them.

Mercury Worker performs the packet-level priority scheduling without throughput penalty. When a local gradient tensor produced by Worker Computing Engine is delivered to Mercury Worker, it is subdivided into Mercury packets. Mercury Worker assigns a priority to every Mercury packet according to the tensor's layer number. The packets with the highest priority will always be sent first. Mercury Worker also receives packets from Mercury PS. These packets are merged to form global gradient tensors that will be fed to Worker Computing Engine afterwards.

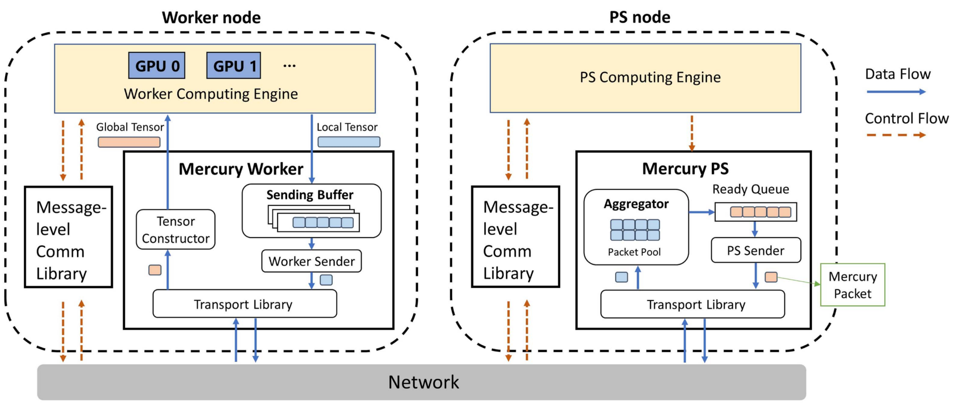


Fig. 8. System architecture.

Mercury PS performs the immediate packet-level gradient aggregation. As described before in *Key idea 3*, the aggregation happens inside Mercury PS. PS Computing Engine does not conduct any computation to gradients. This is necessary to reduce the processing delay of tensors at the packet granularity.

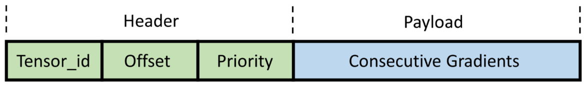


Fig. 9. Mercury packet structure.

C. System Components

Mercury Worker. As shown in Fig. 8, Mercury Worker consists of four modules: 1) *Sending Buffer* that contains multiple Mercury packet queues; 2) *Worker Sender* that extracts packets one by one from *Sending Buffer*; 3) *Transport Library* where the real transmission happens; 4) *Tensor Constructor* that collects received Mercury packets and constructs gradient tensors. Mercury Worker provides push and pull transmission service for deep learning frameworks.

The number of queues in *Sending Buffer* equals the number of supported scheduling priorities. Usually, the number of packets is much larger than that of scheduling priorities. To reduce the time complexity of extracting a packet, we use a group of FIFO queues instead of a single heap priority queue. Each FIFO queue has a unique priority. When a tensor is passed to *Sending Buffer*, it is divided into Mercury packets stored in a FIFO queue corresponding to its priority. In this way, the time complexity of extracting a packet with the highest priority is $O(N)$ where N is the number of queues.

Before the training begins, the user code notifies *Sending Buffer* to initialize the number of Mercury packet queues. We find that most DNN models have a small number of tensors (no more than 260 for our tested models). Therefore, we assign every tensor with a Mercury packet queue in our implementation. An assignment algorithm can be developed to balance the trade-off between the scheduling efficiency and the time complexity when the DNN model has too many tensors.

The structure of a Mercury packet is shown in Fig. 9. The Mercury packet header indicates its address (tensor ID, offset) and priority. The payload of each Mercury packet contains consecutive gradients in one tensor. No gradient is split across two packets and no packet is across two tensors. If we use 3 long integers (i.e., 24B) for the Mercury header, the header is only $\sim$ 0.1% the size of the Mercury packet (e.g., 30 KB). The overhead of the header can be ignored. To avoid the memory copy, each Mercury packet in the FIFO queue of *Sending Buffer* contains only a header other than real data.

Worker Sender constantly extracts Mercury packets (i.e., only the headers) with the highest priority from *Sending Buffer*. Then *Worker Sender* finds data from the original tensor and transmits the header and data to the corresponding PS using the transport library. *Transport Library* executes multiplexing and demultiplexing operations. It encapsulates a Mercury packet to multiple transport library packets (e.g., TCP packets) for sending data and does the reversed procedure while receiving data. Memory copy only happens inside the transport library just as we use the transport library directly without Mercury. It's better to set the Mercury packet size as $n \times S_{pkt}$ with S_{pkt} as the size of a transport library packet (e.g., TCP maximum segment size). Using n as 1 increases the burden of the scheduling system. Here, n is set to 10 $\sim$ 20 in our implementation.

Mercury PS. As shown in Fig. 8, Mercury PS consists of four modules: 1) *Transport Library* that works the same way as in Mercury Worker; 2) *Aggregator* that aggregates gradient packets and buffers unready packets; 3) *Ready Queue* that stores ready gradient packets (i.e., global gradient packets); 4) *PS Sender* that transmits global gradient packets to all workers.

When receiving a Mercury packet, *Aggregator* checks its header and adds gradient values to the corresponding packet (i.e., the packet with the same header) in *Packet Pool*. If no corresponding packet is found, the received packet will be put into *Packet Pool*. If a packet in *Packet Pool* is added for M times (M is the number of workers), it will be pushed to *Ready Queue*. *PS Sender* constantly extracts packets from *Ready Queue*, constructs M copies for each ready packet and transmits them to all workers respectively.

Aggregator, *Ready Queue* and *PS Sender* process Mercury packets in a pipelining manner. Summing up gradients is fast in CPUs. We measure the performance of our *Aggregator* implementation in one of our physical servers. The highest processing speed is $\sim$ 2600 Gbps for FP32 precision, much higher than the widely-used NIC speed. Therefore, the packet-level aggregation will not be a bottleneck. The size of a Mercury packet is often much smaller than that of a tensor. Therefore, our design of Mercury PS obtains the near-perfect overlapping of uplink and downlink transmissions. From the worker's perspective, *push* and *pull* are fully overlapped for each layer. We will show that this design does not impair bandwidth utilization later on.

D. Implementation

We implement Mercury based on TCP using C++. Mercury is integrated with MXNet, a deep learning framework supporting an efficient PS architecture. The implementation includes ~3000 LoC in total, among which ~2700 LoC is written in C++ for Mercury and the rest is python plugins for MXNet. MXNet utilizes *ps-lite* [22] as its underlying PS architecture library, which is in charge of connection establishment and data transmission. The main modification is based on the component *Van*, an abstraction of point-to-point communication in *ps-lite*. The default transmission service in *Van* is based on ZeroMQ. In the current implementation of *ps-lite*, the push, pull and some control functions call communication APIs in *Van* to transmit tensors or messages. We derive two components, namely *ZMQ_Van* and *Mercury_Van*. The former adopts the default transmission implementation to transmit control messages, and the latter transmits tensors. *Mercury_Van* works as Mercury Worker for worker nodes, or Mercury PS for PS nodes. We make no modifications to the code of iterative training and all APIs in the communication stack remain the same as before. We add plugins to the user code of worker. One of them will notify *Mercury_Van* the number of tensors (i.e., the number of scheduling priorities) at the beginning of the training. We also add plugins to the user code of PS to conduct necessary initializations for *Mercury_Van*.

It is easy to integrate Mercury with other deep learning frameworks supporting the PS architecture, such as TensorFlow and PyTorch. Like *ps-lite*, their PS architecture implementations maintain the communication relations of all nodes and are usually built on top of point-to-point communication libraries. To implement Mercury for these frameworks, we only need to add Mercury as one of their point-to-point communication libraries, just like the modification for *ps-lite*. All-reduce architecture [23] can also benefit from the priority scheduling of Mercury.

V. Model and Trace-Driven Analysis

In this section, we model the time usage of one iteration in distributed DNN training under scheduling policies. The potential benefits of Mercury are illustrated by conducting trace-driven analysis using our models.

TABLE I
NOTATIONS

Name	Discription
B	Network bandwidth of all workers and PSes
L	The number of layers in DNN model, i.e., the number of tensors
S^l	The size of tensor of layer l
bp^l	The backward propagation of layer l
fp^l	The forward propagation of layer l
upd^l	The optimizer/updater operation of layer l
$push^l$	The push operation of layer l
$pull^l$	The pull operation of layer l
t^l_{bp}	Backward propagation time of layer l
t^l_{fp}	Forward propagation time of layer l
t^l_{upd}	Optimizer/updater execution time of layer l
t^l_{push}	Push operation time of layer l
t^l_{pull}	Pull operation time of layer l
t_{iter}	Time of one iteration in distributed training
τ^l_{bp}	Timestamp when backward propagation of layer l finishes
τ^l_{fp}	Timestamp when forward propagation of layer l finishes
τ^l_{push}	Timestamp when push operation of layer l finishes
τ^l_{pull}	Timestamp when pull operation of layer l finishes

A. Modeling Iteration Time

We focus on the paradigm that the optimizer (e.g., SGD, Adam) is executed in workers. For the convenience of modeling, we assume that:

1) All computation operations form a chain (i.e., no computation operations are executed in parallel in one GPU).
2) There is no hardware heterogeneity (GPUs, CPUs and NICs) across workers and PSes.
3) The aggregation is instantaneous.

Assumption 1 is typically true for deep learning frameworks and most DNN models. Some special models like Transformer have a part of complicated dataflow subgraph. The model as a whole can be treated as a chain in general. With Assumption 2, a PS receives the same gradient from all workers at the same time. Assumption 3 indicates that the aggregation time is ignored. Simply summing local gradients is much faster than other computations.

The frequently used variables are listed in Table I. The model takes $B, L, S^l, t^l_{bp}, t^l_{fp}$ and t^l_{upd} as inputs. We regard bp^L as the start of one iteration and fp^L as the end. We denote the start timestamp of bp^L as 0. We first present general equations that will be used later on.

The push and pull operation of layer l are calculated as follows

$$t^l_{push} = \frac{S^l}{B}. \tag{1}$$

$$t^l_{pull} = \frac{S^l}{B}. \tag{2}$$

Backward propagations are independent of communication, so the timestamp of finishing bp^l is

$$\tau^l_{bp} = \sum\nolimits_{i=l}^{L} t^i_{bp}. \tag{3}$$

Algorithm 1: Simulated Scheduling of Mercury.

```
   Input: $\boldsymbol{\tau}_{bp}$ , $\boldsymbol{t}_{bp}$ , $\boldsymbol{t}_{push}$
   Output: $\boldsymbol{\tau}_{push}$
1  $\tau_{now} = 0$ , $Buf = \emptyset$
2  for $l = 1 \rightarrow L$ do
3      $\hat{\boldsymbol{t}}_{push}[l] = \boldsymbol{t}_{push}[l]$
4  for $l = L \rightarrow 1$ do
5      $\tau_{now} = \boldsymbol{\tau}_{bp}[l]$
6      $Buf = Buf \cup \{l\}$
7      if $l = 1$ then
8          $Buf$ , $\hat{\boldsymbol{t}}_{push}$ , $\boldsymbol{\tau}_{push}$ = CONSUME($Buf$ , $\hat{\boldsymbol{t}}_{push}$
            , $\boldsymbol{\tau}_{push}$ , $\tau_{now}$ , $+\infty$ )
9      else
10         $Buf$ , $\hat{\boldsymbol{t}}_{push}$ , $\boldsymbol{\tau}_{push}$ = CONSUME($Buf$, $\hat{\boldsymbol{t}}_{push}$,
            $\boldsymbol{\tau}_{push}$ , $\tau_{now}$ , $\boldsymbol{t}_{bp}[l-1]$ )
11 return $\boldsymbol{\tau}_{push}$
12 Function CONSUME ($Buf$ , $\hat{\boldsymbol{t}}_{push}$ , $\boldsymbol{\tau}_{push}$, $\tau_{now}$ ,
    $\Delta t$):
13     while $\Delta t > 0$ and $Buf \neq \emptyset$ do
14         $l = \min_{i \in Buf} i$
15         if $\Delta t \geq \hat{\boldsymbol{t}}_{push}[l]$ then
16             $\Delta t = \Delta t - \hat{\boldsymbol{t}}_{push}[l]$
17             $\tau_{now} = \tau_{now} + \hat{\boldsymbol{t}}_{push}[l]$
18             $\hat{\boldsymbol{t}}_{push}[l] = 0$
19             $Buf = Buf / \{l\}$
20             $\boldsymbol{\tau}_{push}[l] = \tau_{now}$
21         else
22             $\hat{\boldsymbol{t}}_{push}[l] = \hat{\boldsymbol{t}}_{push}[l] - \Delta t$
23             $\Delta t = 0$
24     return $Buf$ , $\hat{\boldsymbol{t}}_{push}$ , $\boldsymbol{\tau}_{push}$
```

The forward propagation of layer l relies on fp^{l-1} and upd^l, while upd^l relies on $pull^l$. In real training, fp^{l-1} and upd^l can both be scheduled ahead of another one. The optimizer's execution time is usually much smaller than the forward propagation. We focus on communication scheduling but not computation scheduling. To simplify the model, we assume that upd^l is always executed after fp^{l-1}. Based on the above assumption, we can equivalently redefine the prerequisites of fp^l as: 1) fp^l relies on upd^l; 2) upd^l relies on fp^{l-1} and $pull^l$. Therefore, the timestamp of finishing fp^l is

$$\tau_{fp}^l = \begin{cases} \tau_{pull}^l + t_{upd}^l + t_{fp}^l, & l = 1; \\ \max\{\tau_{pull}^l, \tau_{fp}^{l-1}\} + t_{upd}^l + t_{fp}^l, & 2 \leq l \leq L. \end{cases} \tag{4}$$

Finally, the iteration time can be represented by

$$t_{iter} = \tau_{fp}^L. \tag{5}$$

Based on the above equations, we can see that t_{iter} is mainly decided by τ_{pull}^l. Different scheduling policies have different communication patterns, producing different τ_{pull}^l. We model τ_{pull}^l under scheduling policies as below.

B. Modeling Scheduling Policies

WFBP. As shown in Fig. 2(a), the two prerequisites for the push operation of layer l are: 1) the BP of layer l has finished; 2) the push operation of layer $l+1$ has finished. Therefore, the timestamp of finishing $push^l$ is

$$\tau_{push}^l = \begin{cases} \tau_{bp}^l + t_{push}^l, & l = L; \\ \max\{\tau_{bp}^l, \tau_{push}^{l+1}\} + t_{push}^l, & 1 \leq l \leq L-1. \end{cases} \tag{6}$$

The aggregation time is ignored since simply summing up local gradients is fast. Thus, the timestamp to finish $pull^l$ is

$$\tau_{pull}^l = \tau_{push}^l + t_{pull}^l. \tag{7}$$

Mercury. We assume that the Mercury packet size is infinitesimal since most tensors are overwhelmingly larger than a Mercury packet. For example, a tensor in VGG-16 is ~400 MB, while Mercury packet size is ~16 KB. This assumption indicates that: 1) preemption happens immediately; 2) push and pull are fully overlapped. Then, we have

$$\tau_{pull}^l = \tau_{push}^l. \tag{8}$$

Priority scheduling and immediate preemption make it hard to express τ_{push}^l as a close-form function. Therefore, to calculate τ_{push}^l with Mercury, we introduce Algorithm 1 running in a dynamic manner to simulate the scheduling of push operations. Buf is an abstraction of tensor buffer to store tensors ready to be "pushed". $\hat{\boldsymbol{t}}_{push}[l]$ is the remaining time needed to finish the push operation of layer l at a certain timestamp. It equals 0 after the push operation finishes. The algorithm first initializes Buf and $\hat{\boldsymbol{t}}_{push}$ (Line 1-3), which are updated dynamically. τ_{now} is the current timestamp. Line 4-10 simulates the events after the BP of layer l is finished. Since the tensor of layer l is ready, the algorithm adds l into Buf (Line 6). Then the time period $\boldsymbol{t}_{bp}[l-1]$ (i.e., t_{bp}^{l-1}) is consumed to transmit tensors (Line 10).

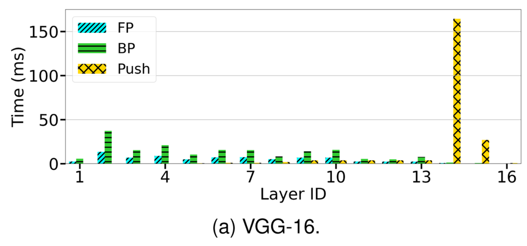
In function *CONSUME*, the tensor with a smaller layer number has a higher priority to consume the time frame (Line 14). If a push operation is finished, the layer number will be excluded from Buf (Line 19) and the finish timestamp will be recorded (Line 20).

Theorem 1. Mercury obtains the minimum iteration time, given the following assumptions:

1) Mercury packet size is infinitely small.
2) All computation operations form a chain (i.e., no parallel computation operations).
3) The DNN engine operates at the work-conserving mode (i.e., tensor fusion of nearby layers is not incorporated).
4) The aggregation is instantaneous.
5) The optimizer is executed in workers.

Theorem 1 shows that Mercury turns out to be optimal with ideal assumptions. It can be proved by induction on τ_{fp}^l. The detailed proof is provided in the Appendix, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2023.3250462.

Mercury W/O Preemption. We disable priority scheduling for breakdown analysis. Specifically, Mercury packets are transmitted in a FIFO way. In this way, preemption in Mercury Worker is not allowed but Mercury PS can still perform the immediate



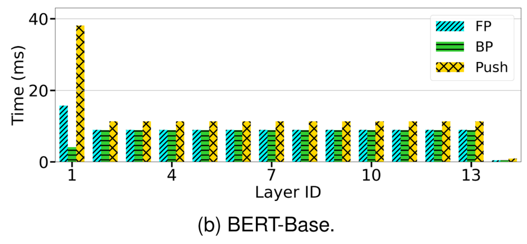


Fig. 10. Layered time usage with NVIDIA GTX 3090 24 GB GPU supposing the network bandwidth is 20 Gbps.

packet-level gradient aggregation. Mercury W/O Preemption can only benefit from push and pull overlapping. As for the model implementation, we only need to change Line 14 in Algorithm 1. For Mercury W/O Preemption, l should be popped out of Buf in a FIFO way.

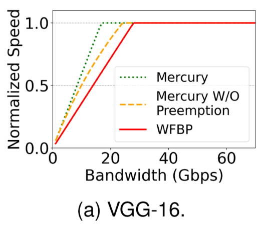
Oracle. We define t_{oracle} as the iteration time of training with one GPU locally. Then, we have

$$t_{oracle} = \sum_{i=l}^{L} (t_{bp}^{i} + t_{upd}^{i} + t_{fp}^{i}). \quad (9)$$

t_{oracle} is always less or equal to t_{iter} under any communication scheduling policies. If a scheduling policy can hide all communication time by the computation time, its t_{iter} is equal to t_{oracle}.

C. Trace-Driven Analysis

To illustrate how the distributed training may benefit from the scheduling policies, we analyze the theoretical performance gain based on our models. Fig. 10 shows the distribution of computation (BP, FP) time and communication time of different layers of VGG-16 and BERT-Base. The communication time is calculated by dividing the size of each layer's tensor by the bandwidth (20 Gbps). The computation time is measured when training locally using a NVIDIA GTX 3090 24 GB GPU. BERT-Base has too many layers. For the illustration in Fig. 10(b), we merge these layers into simple 14 layers. Taking above measured results as inputs to our models, we show the normalized speed in Fig. 11. The normalized speed is defined by the ratio of t_{oracle} to t_{iter} of different scheduling policies. We can see that Mercury is superior to WFBP over a large bandwidth range. BERT-Base has large tensors in the first few layers (i.e., embedding layers). So it is hard for BERT-Base to reach the best scalability even with high bandwidth. VGG-16 and most other classification DNN models have large tensors in the last few layers (i.e., fully connected layers), making it easier to reach the best scalability. Fig. 12 shows the minimum required



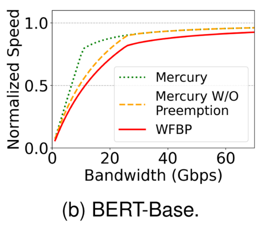


Fig. 11. The potential scheduling efficiency of schedulers with NVIDIA GTX 3090 24 GB GPU.

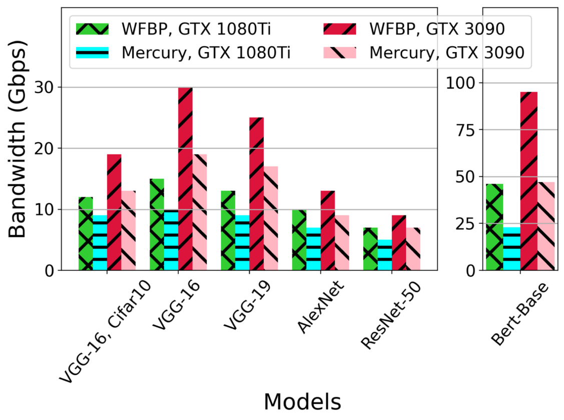


Fig. 12. Minimum required bandwidth to reach the best scalability of different schedulers with different GPUs. The smaller, the better. .

TABLE II
MODEL DETAILS IN EXPERIMENTS

Model	Dataset	# of parameters (million)	Batch size	Input size
VGG-16	Cifar10	~ 35	128	$3 \times 32 \times 32$
VGG-16	Caltech101	~ 135	32	$3 \times 224 \times 224$
VGG-19	Caltech101	~ 140	32	$3 \times 224 \times 224$
AlexNet	Caltech101	~ 57	512	$3 \times 224 \times 224$
ResNet-50	Caltech101	~ 24	32	$3 \times 224 \times 224$
BERT-Base	Caltech101	~ 109	32	128

bandwidth to reach the best scalability. The minimum required bandwidth is defined as the minimum bandwidth to obtain $t_{oracle}/t_{iter} > 0.90$ for BERT-Base or $t_{oracle}/t_{iter} > 0.99$ for other DNN models. The model configurations can be found in Table II. In general, Mercury reduces the required bandwidth to reach the best scalability by up to $\sim$ 50%. A faster GPU requires higher bandwidth to obtain good scalability. In today's GPU clusters, the training may be conducted on GPUs faster than NVIDIA GTX 3090 GPU. Besides, the available bandwidth is usually shared by multiple distributed machine learning jobs in multi-tenant clusters. Therefore, it is beneficial to deploy Mercury for distributed training jobs in GPU clusters.

VI. Evaluation

In this section, we evaluate the acceleration ratio of Mercury on our cluster using the distributed training workloads of popular DNN models.

A. Methodology

Testbed Setup. Our testbed has eight physical servers. Each server has one 8-core Intel Xeon W2140b CPU at 3.20 GHz and 32 GB of DDR4 RAM. Four servers are equipped with one NVIDIA GTX 1080Ti 11 GB GPU as worker nodes, while the other four servers will run as PS nodes. Each server is connected to the switch using one Mellanox ConnectX-3 NIC, providing a 10 Gbps Ethernet. The OS is Ubuntu 18.04 with Linux kernel 5.4. The deep learning framework is MXNet-1.5.0 with CUDA-9.0 and cuDNN-7.1.3.

When DNN models are trained with only GPU machines and no extra CPU machines are provided, PSes can only be colocated with workers. Thus in one iteration, each GPU machine sends (resp. receives) $2(n-1)S/n$ units of traffic to (resp. from) the network with S as the model size and n as the number of workers. This is the same traffic volume as that of training with the all-reduce architecture. So the training speed is similar under these two architectures. This training scenario degrades the potential benefits of PS architecture. Prior work [18] shows that when there are extra CPU machines for PS architecture, the bandwidth is utilized twice more efficiently than all-reduce. Large-scale clusters [24], [25] host an enormous pool of CPUs and network bandwidth apart from GPUs. GPUs are extremely expensive compared with CPUs and network bandwidth. For example, using AWS, renting GPU machines costs $\sim$100 times the price of renting CPU machines. For training with PS architecture, just a little more spending can bring up to 100% improvement. Therefore, in our experiments, we focus on the scenario where CPU machines (i.e., PSes) are as many as GPU machines (i.e., workers).

For communication of multiple GPUs in the same machine, PCIe links and NVLink are less likely the communication bottleneck than NIC. We focus on the scheduling of inter-machine communication in our experiments.

Baselines. We compare Mercury with the following communication scheduling methods.

1) *Linear-Scaling:* This pseudo approach calculates the ideal speedup [13], [16], [18] that multiplies the training speed at a single worker by the number of workers.
2) *Native-PS:* MXNet adopts WFBP as its default scheduling method for PS architecture.
3) *ByteScheduler:* ByteScheduler is the state-of-the-art user-level tensor partitioning method. The partition size and credit size are tuned automatically for better performance.
4) *BytePS:* For inter-machine communication, BytePS develops a traffic load assignment strategy to prevent a bottleneck node from slowing down the training system. It incorporates the idea of user-level tensor partitioning and priority scheduling for better pipelining, but hyper-parameters for tensor partitioning are fixed.
5) *Mercury W/O Preemption:* This method is introduced in Section V-B. Our purpose is to analyze the contribution of each Mercury component to the acceleration ratio.

Benchmark DNN Models. We evaluate Mercury's performance with four widely used image classification models: VGG-16, VGG-19, AlexNet [26] and ResNet-50, and one language processing Transformer model, BERT-Base [2]. The details of the training models are shown in Table II. The batch size is the number of images or sentences processed at each iteration per GPU. The input size is the resolution of input images for CNN models, and is the length of input sentences for Transformer models. Workers and PSes are on different physical servers, and the number of workers is equal to the number of PSes.

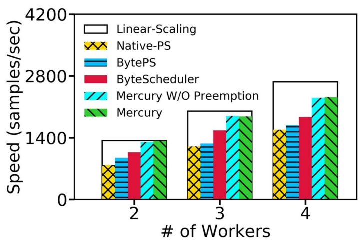
Metrics. We adopt the training speed (samples per second) as the main performance metric. All the speed magnitudes are measured over 100 iterations after a warm-up. The warm-up is long enough for ByteScheduler's tuning algorithm to find proper hyper-parameters. We also measure network throughput while training.

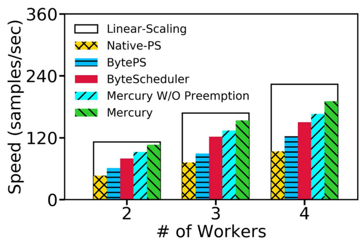
B. Optimization Under Static Resources

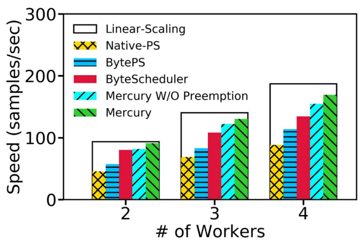
Speedup of Different DNN Models. We evaluate the performance improvements achieved by Mercury when training with different numbers of workers. Fig. 13 shows the training speeds of different schemes under the 10 Gbps network with the number of workers ranging from 2 to 4. In general, Mercury delivers the best performance for all benchmark DNN models. Mercury outperforms Native-PS, BytePS and ByteScheduler by up to 130%, 74% and 40% respectively. For breakdown analysis, Mercury outperforms Mercury W/O Preemption by up to 15%. We also find that Mercury achieves better speedups in VGG-16 and VGG-19 than in AlexNet and BERT-Base. This is because of their different ratios of communication time over computation time. From Fig. 13(e), we observe that Mercury has a very gentle speedup for ResNet-50. The reason is that the 10 Gbps bandwidth is enough for ResNet-50 with WFBP. It is consistent with Fig. 12 in Section V-C.

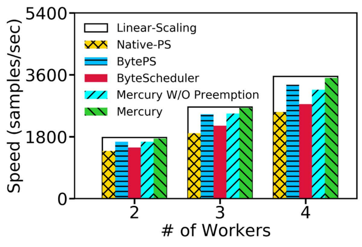
Deep Dive Into Mercury. We measure the worker's uplink and downlink throughput when training VGG-16 and the results are shown in Fig. 14. We also illustrate the execution time of BP, FP and push time at an iteration in Fig. 15 by using MXNet Profiler. One can see that Mercury can hide all communication time and the GPU is always busy, while some idle time appears in Mercury W/O Preemption. In Fig. 14, the speedup of Mercury W/O Preemption over Native-PS mainly comes from the overlap of uplink and downlink transmissions. The speedup of Mercury over Mercury W/O Preemption comes from the better overlap of computation and communication. The network bandwidth is fully utilized by both Mercury and Mercury W/O Preemption, indicating that our design eliminates unnecessary startup time.

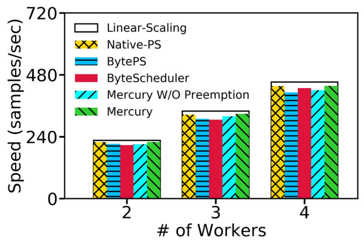
We show the CPU utilization of different schedulers in Fig. 16. The CPU utilization is calculated by multiplying the iteration time and the CPU utilization per second. Then it's normalized by the value of Native-PS. User-level tensor partitioning methods consume more CPU resources than Mercury. Compared to Native-PS, Mercury speeds up the training by up to 104% with


(a) VGG-16, Cifar10.
$(38\% - 48\%)$, $(20\% - 24\%)$


(b) VGG-16.
$(55\% - 74\%)$, $(27\% - 34\%)$


(c) VGG-19.
$(49\% - 59\%)$, $(13\% - 26\%)$


(d) AlexNet.
$(5\% - 7\%)$, $(18\% - 28\%)$


(e) ResNet-50.
$(3\% - 6\%)$, $(2\% - 7\%)$

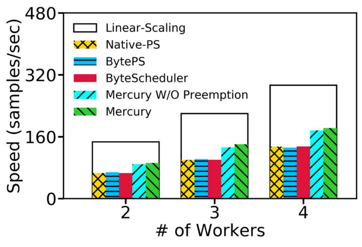
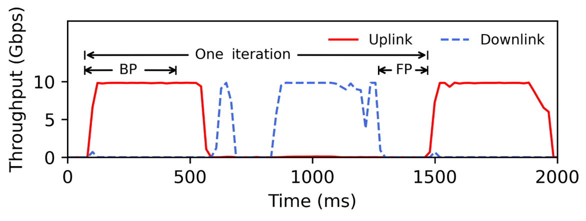
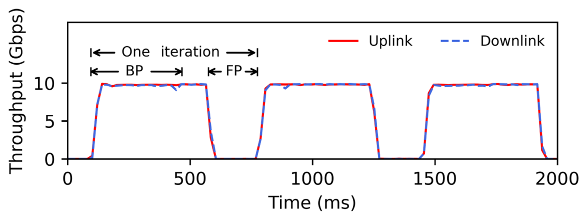

(f) BERT-Base.
$(35\% - 39\%)$, $(35\% - 40\%)$

Fig. 13. Training DNN models under the 10 Gbps network. The numbers in the first (second) parentheses are speedup percentages of Mercury over BytePS (ByteScheduler).


(a) Native-PS.


(b) Mercury W/O Preemption.

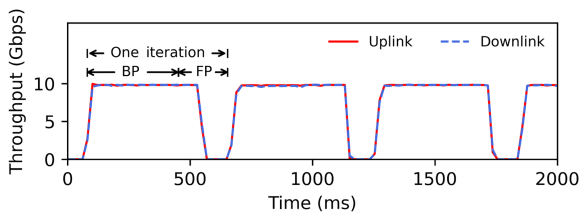
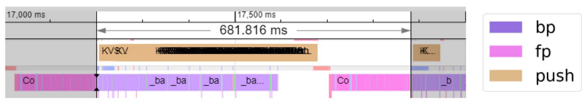

(c) Mercury.

Fig. 14. Training VGG-16 with 2 workers and 2 PSes using MXNet. The network bandwidth is 10 Gbps.


(a) Mercury W/O Preemption.

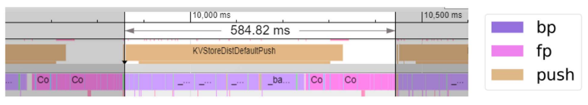

(b) Mercury.

Fig. 15. Execution timeline of one iteration in MXNet.

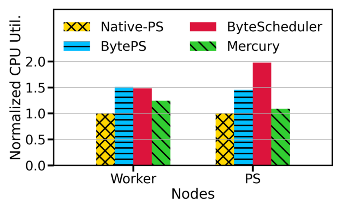


Fig. 16. Normalized CPU utilization per iteration when training VGG-16 with 4 workers and 4 PSes.

the cost of only 23% (and 9%) extra CPU computing resources on workers (and PSes).

C. Optimization Under Dynamic Resources

Speedup of Different DNN Models. Mercury can further speed up the training in a dynamic environment. We limit the maximum bandwidth of servers dynamically using the Linux WonderShaper tool. Figs. 18 and 19 show the performance

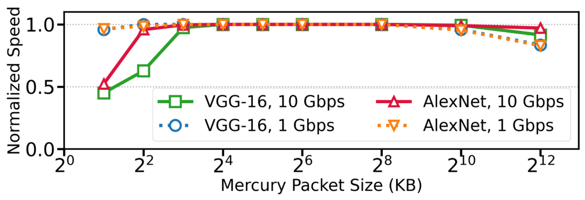


Fig. 17. The effect of Mercury packet size on the training speed when training VGG-16 and AlexNet with 2 workers and 2 PSes.

under two dynamic scenarios. In Fig. 18, Mercury outperforms Native-PS, BytePS and ByteScheduler by up to 85%, 74% and 41% respectively. In Fig. 19, Mercury outperforms Native-PS, BytePS and ByteScheduler by up to 129%, 99% and 104% respectively. We can see that Mercury's speedup over the user-level tensor partitioning methods is higher than that in static network environments. For ResNet-50, Mercury can accelerate its training speed by up to 20% in these scenarios. The user-level tensor partitioning methods are hard to adapt to highly dynamic changing bandwidth since the well-tuned hyper-parameters in one bandwidth setting may operate poorly in another setting. For example, ByteScheduler (BytePS) outperforms Native-PS by up to 76.5% (79.8%) in Fig. 13(c), while only 7.3% (9.9%) in Fig. 19(c).

Hyper-Parameter Tuning Analysis. Mercury is robust to the changing bandwidth. We measure how the Mercury packet size affects the training performance. Fig. 17 shows the results of a big model VGG-16 and a small model AlexNet. Other DNN models have similar results. Mercury has suboptimal performance only when the packet size is supper small (i.e., $\leq$ 8 KB) or super big (i.e., $\geq$ 1 MB). The former causes bandwidth underutilization and the latter leads to big preemption granularity for tensor scheduling. When the packet size is between 16 KB to 256 KB, the training speed is always optimal for different bandwidth settings. In our implementation, the Mercury packet size is fixed to $\sim$ 30 KB. In ByteScheduler's experiments for these scenarios, its "optimal" partition size is $1 \sim 10$ MB. The big partition size undermines the scheduling efficiency of ByteScheduler.

Multiple Jobs Sharing the Cluster. We further evaluate the training speed when multiple distributed training jobs share the computation and communication resources. In each scenario of Fig. 20, the same 4 workers and 4 PSes are utilized for all jobs. The training speed is normalized by that of Native-PS. In Fig. 20(a), (b), and (c), we set the batch size of VGG-16, VGG-19 and BERT-Base to 16 because of the GPU memory limitation. In general, Mercury outperforms other methods. In Fig. 20(b), Mercury outperforms Native-PS, BytePS and ByteScheduler by up to $\sim$ 100%, $\sim$ 38% and $\sim$ 48% for both jobs. In Fig. 20(d), Mercury only improves the training speed of VGG-16 with Cifar10 and does not harm the speed of the other 2 jobs. ByteScheduler obtains a higher speed than Mercury for AlexNet, but with the cost of reducing the speed of the other 2 jobs significantly. It is interesting that the job with a higher bandwidth requirement (e.g., VGG-16 in Fig. 20(a) or VGG-16 with Cifar10 in Fig. 20(d)) may benefit more from Mercury in a shared cluster. We leave it as future work to combine Mercury with job scheduling.

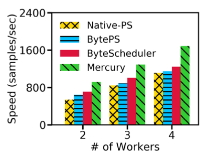
D. Optimization Under Heterogeneous Resources

Mercury can also be applied to heterogeneous training systems. With heterogeneous GPU workers like Fig. 21(a), gradient tensors in faster GPU workers are produced and transmitted to PSes earlier. The bottleneck is the computation and communication of tensors in slow workers. Mercury still performs better on scheduling the communication of slow workers than other schedulers. In Fig. 21(a), Mercury outperforms Native-PS and BytePS by up to 93% and 37% for VGG-19. For BERT-Base, Mercury outperforms Native-PS and BytePS by up to 26% and 14% respectively. For ResNet-50, all schedulers achieve no better performance than Native-PS. Because 10 Gbps is enough for WFBP to reach the best scalability on ResNet-50 with both types of GPU, as shown in Fig. 12. When training with no-balanced links, the bottleneck is the communication in links with heavy traffic. Compared to other schedulers, Mercury still performs better on scheduling the communication of bottleneck links. In Fig. 21(b), workers and PSes are not in a perfect load balancing. Obviously, Mercury outperforms other methods for all DNN models.

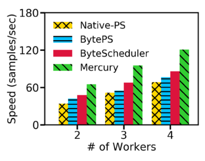
VII. Related Work

DNN Training Frameworks. To facilitate the computation operations (e.g., forward propagation, backward propagation), DNN compilers and libraries have been proposed from both industry and academia, including MKL [27], cuDNN [28], XLA [29], TVM [30] and TensorRT [31].

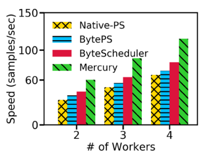
Communication Scheduling. Communication scheduling approaches accelerate distributed DNN training by overlapping computation and communication. Poseidon [13] exploited WFBP to hide the communication time by backward propagations. TicTac [14] delivered the optimal transmission order of DNN tensors through critical-path analysis on the dependency graph. Tensor partitioning is an important technique to improve the efficiency of communication scheduling. P3 [15] sliced tensors into small pieces to perform fine-grained priority scheduling. ByteScheduler [16] proposed a Bayesian optimization approach to tune hyper-parameters of tensor partitioning for better performance across different system configurations. AutoByte [17] presented a real-time method that searches the optimal hyper-parameters as the training environments dynamically change. AutoByte used a meta-network to predict speedups under specific configurations. Considering the internal topology of a multiple-GPU machine, BytePS [18] developed an optimal intra-machine communication strategy and adopts tensor partitioning for better pipelining. Tensor fusion [32], [33], [34] reduced communication time by merging nearby small tensors into a large one. ASC-WFBP [34] leveraged simultaneous communications to extend the design space of tensor fusion. iPart [35] proposed to partition the sequential execution of communication and computation with various sizes. We proposed a transport layer based communication scheduling approach [36].



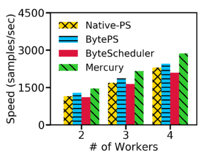
(a) VGG-16, Cifar10.
(43% – 47%)
(28% – 36%)



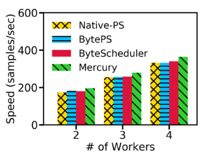
(b) VGG-16.
(55% – 74%)
(36% – 41%)



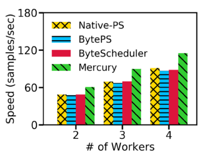
(c) VGG-19.
(50% – 59%)
(35% – 40%)



(d) AlexNet.
(14% – 17%)
(31% – 37%)

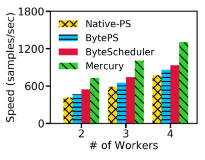


(e) ResNet-50.
(8% – 10%)
(7% – 8%)

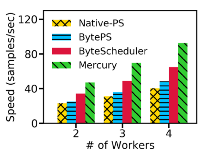


(f) BERT-Base.
(27% – 33%)
(25% – 30%)

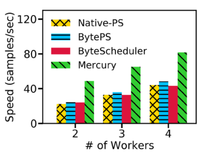
Fig. 18. Training DNN models when the network bandwidth varies randomly between 6 Gbps and 10 Gbps every 5 seconds. The numbers in the first (second) parentheses are speedup percentages of Mercury over BytePS (ByteScheduler).



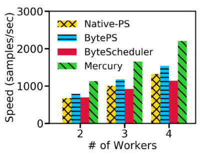
(a) VGG-16, Cifar10.
(51% – 56%)
(34% – 46%)



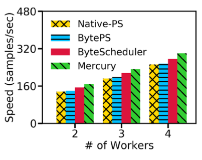
(b) VGG-16.
(88% – 95%)
(37% – 43%)



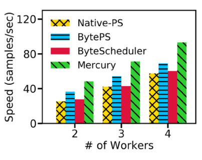
(c) VGG-19.
(69% – 99%)
(89% – 104%)



(d) AlexNet.
(41% – 43%)
(63% – 93%)

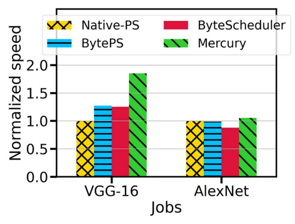


(e) ResNet-50.
(17% – 20%)
(7% – 9%)

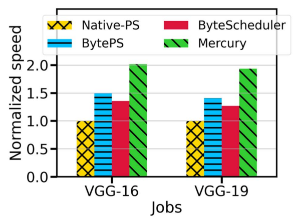


(f) BERT-Base.
(31% – 35%)
(55% – 75%)

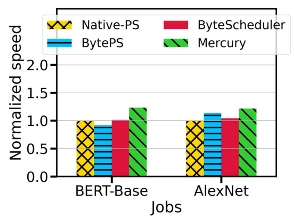
Fig. 19. Training DNN models when the network bandwidth varies randomly between 1 Gbps and 10 Gbps every 5 seconds.



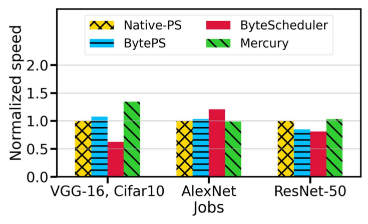
(a)



(b)

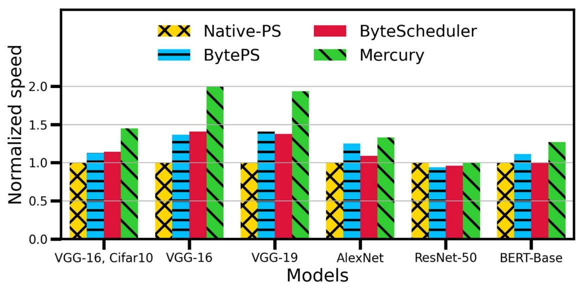


(c)

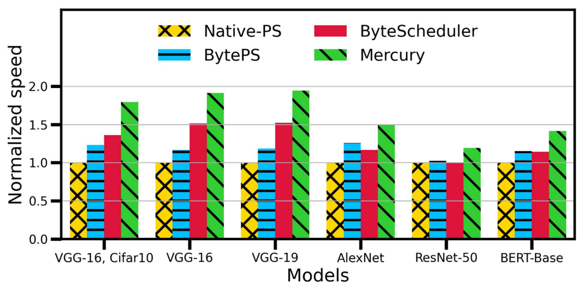


(d)

Fig. 20. Four scenarios of training multiple DNN models simultaneously.



(a) Training with 4 workers and 4 PSes. Two workers are equipped with one NVIDIA GTX 1080Ti 11 GB GPU and the other two are equipped with one NVIDIA GTX 3090 24 GB GPU.



(b) Training with 4 workers and 2 PSes.

Fig. 21. Training models (a) with heterogeneous GPU workers and (b) without load balancing.

Communication Compression. Compressing the gradients or model parameters before transmission is an effective approach to reduce the communication overhead of distributed DNN training. There exist two popular compression methods, *quantization* [37], [38], [39] and *sparsification* [40], [41], [42]. Quantization reduces the precision of each variable by limiting the number of bits or the code-book mapping. Sparsification reduces the number of transmitted variables by selecting only relatively important ones. In terms of compression effectiveness, the sparsification approaches are more effective especially at the late stage of training since most of the gradients are sufficiently close to zero. Sparsification and quantization are two orthogonal approaches that are superimposed to further improve the compression capability [43], [44], [45]. Generally, these compression techniques can work with Mercury in parallel.

Alternative Communication Structures. BSP pursues the perfect model synchronization in every training round. In realistic clusters, the GPU load and the traffic volume of a worker can be time-varying, and the computing powers of different GPUs are usually different. Hence, the fast workers have to wait for the slow ones that are usually deemed as *stragglers*. Asynchronous Parallel (ASP) [46] allowed a worker to proceed the model training without waiting for the arrival of the synchronized global model. Stale Synchronous Parallel (SSP) [47], [48] introduced a latency threshold to avoid excessive staleness. The fast workers can lead the slow ones by a margin, and have to wait for them if this threshold is reached. DS-Sync [49] presented a novel divide-and-shuffle approach to synchronize parameters in a bottleneck-free manner. Classic ring-based all-reduce cannot adapt to the specific network topology of data center networks (DCN). Some works proposed new all-reduce architectures like hierarchical all-reduce [50] and tree-based all-reduce [51]. As an underlying scheduler, Mercury can potentially accelerate these new communication architectures.

In-Network Solutions. The communication link between workers and PSes is usually the bottleneck of distributed training. The temporary congestion at the network switches might create the straggler problem, and the traffic passing through them are usually redundant. CEFS [52] and Geryon [53] implemented priority scheduling based on RDMA and commodity switches to perform in-network flow scheduling. Multiple flows with different priorities between PSes and workers are used to transfer gradients and parameters separately. In-network aggregation approaches [54], [55], [56], [57] have been proposed to aggregate gradients using programmable switches to avoid intense bursts of traffic among workers. SwitchML [56] presented a rack-scale architecture where a single switch centrally aggregates model updates for associated workers. To deal with heavy contention for limited switch resources across multiple tenants, ATP [57] developed a decentralized, dynamic and best-effort aggregation mechanism, enabling distributed training jobs to fall back to end-host aggregation if necessary.

VIII. Conclusion

In distributed DNN training with data parallelism, tensor partitioning improves the efficiency of communication scheduling. While it brings communication overhead, resulting in low bandwidth utilization and hard hyper-parameter tuning, especially under dynamic environments. In this paper, we propose Mercury, a scheduler to exploit the scheduling ability above the transport layer to improve the communication efficiency. At the worker side, Mercury provides priority scheduling at the packet granularity. At the parameter server side, Mercury supports immediate aggregation at the packet granularity. We formulate theoretic models of iteration time and analyze the potential speedup based on collected traces. Mercury is implemented in MXNet and evaluated on a testbed with eight servers. Extensive experiments show that Mercury significantly outperforms the existing tensor partitioning methods by more efficient scheduling and better bandwidth utilization.

References

[1] D. He, H. Lu, Y. Xia, T. Qin, L. Wang, and T.-Y. Liu, “Decoding with value networks for neural machine translation,” in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 177–186.

[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” in *Proc. Annu. Meeting Assoc. Comput. Linguistics: Hum. Lang. Technol.*, 2019, pp. 4171–4186.

[3] S. Qiao, Z. Zhang, W. Shen, B. Wang, and A. Yuille, “Gradually updated neural networks for large-scale image recognition,” in *Proc. Int. Conf. Mach. Learn.*, 2018, pp. 4188–4197.

[4] D. Amodei et al., “Deep speech 2: End-to-end speech recognition in english and mandarin,” in *Proc. Int. Conf. Mach. Learn.*, 2016, pp. 173–182.

[5] J. Wang, P. Huang, H. Zhao, Z. Zhang, B. Zhao, and D. L. Lee, “Billion-scale commodity embedding for e-commerce recommendation in alibaba,” in *Proc. 24th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2018, pp. 839–848.

[6] S. J. Chen et al., “Improving recommendation quality in Google drive,” in *Proc. 26th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2020, pp. 2900–2908.

[7] M. Li et al., “Scaling distributed machine learning with the parameter server,” in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2014, pp. 583–598.

[8] Z. Zhang, C. Chang, H. Lin, Y. Wang, R. Arora, and X. Jin, “Is network the bottleneck of distributed training?,” in *Proc. Workshop Netw. Meets AI ML*, 2020, pp. 8–13.

[9] S. Shi, Z. Tang, X. Chu, C. Liu, W. Wang, and B. Li, “A quantitative survey of communication optimizations in distributed deep learning,” *IEEE Netw.*, vol. 35, no. 3, pp. 230–237, May/Jun. 2021.

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

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

[12] M. Abadi et al., “TensorFlow: A system for large-scale machine learning,” in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2016, pp. 265–283.

[13] 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.

[14] S. H. Hashemi, S. A. Jyothi, and R. H. Campbell, “TicTac: Accelerating distributed deep learning with communication scheduling,” in *Proc. 2nd SysML Conf.*, 2019, pp. 418–430.

[15] A. Jayarajan, J. Wei, G. Gibson, A. Fedorova, and G. Pekhimenko, “Priority-based parameter propagation for distributed DNN training,” in *Proc. 2nd SysML Conf.*, 2019, pp. 132–145.

[16] 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.

[17] Y. Ma, H. Wang, Y. Zhang, and K. Chen, “Automatic configuration for optimal communication scheduling in DNN training,” 2022, *arXiv:2112.13509*.

[18] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, “A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters,” in *Proc. 14th USENIX Symp. Operating Syst. Des. Implementation*, 2020, pp. 463–479.

[19] L. G. Valiant, “A bridging model for parallel computation,” *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[20] A. G. Howard et al., “MobileNets: Efficient convolutional neural networks for mobile vision applications,” 2017, *arXiv:1704.04861*.

[21] NVIDIA NCCL, 2022. [Online]. Available: https://developer.nvidia.com/nccl

[22] ps-lite, 2022. [Online]. Available: https://github.com/dmlc/ps-lite

[23] A. Sergeev and M. Del Balso, “Horovod: Fast and easy distributed deep learning in tensorflow,” 2018, *arXiv:1802.05799*.

[24] Q. Weng et al., “MLaaS in the wild: Workload analysis and scheduling in large-scale heterogeneous GPU clusters,” in *Proc. 19th USENIX Symp. Networked Syst. Des. Implementation*, 2022, pp. 945–960.

[25] W. Xiao et al., “AntMan: Dynamic scaling on GPU clusters for deep learning,” in *Proc. 14th USENIX Symp. Operating Syst. Des. Implementation*, 2020, pp. 533–548.

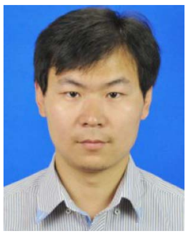
[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.

[27] MKL Intel, 2022. [Online]. Available: https://software.intel.com/en-us/mkl
[28] NVIDIA cuDNN, 2022. [Online]. Available: https://developer.nvidia.com/cudnn
[29] XLA, 2022. [Online]. Available: https://www.tensorflow.org/xla
[30] T. Chen et al., "{TVM }: An automated { End-to-End} optimizing compiler for deep learning," in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 578–594.
[31] NVIDIA TensorRT. [Online]. Available: https://devblogs.nvidia.com/deploying-deep-learning-nvidia-tensorrt/
[32] S. Shi, X. Chu, and B. Li, "MG-WFBP: Efficient data communication for distributed synchronous SGD algorithms," in *Proc. IEEE Conf. Comput. Commun.*, 2019, pp. 172–180.
[33] S. Shi, X. Chu, and B. Li, "MG-WFBP: Merging gradients wisely for efficient communication in distributed deep learning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 8, pp. 1903–1917, Aug. 2021.
[34] S. Shi, X. Chu, and B. Li, "Exploiting simultaneous communications to accelerate data parallel distributed deep learning," in *Proc. IEEE Conf. Comput. Commun.*, 2021, pp. 1–10.
[35] S. Wang, A. Pi, X. Zhou, J. Wang, and C.-Z. Xu, "Overlapping communication with computation in parameter server for scalable DL training," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 9, pp. 2144–2159, Sep. 2021.
[36] Q. Duan, Z. Wang, Y. Xu, S. Liu, and J. Wu, "Mercury: A simple transport layer scheduler to accelerate distributed DNN training," in *Proc. IEEE Conf. Comput. Commun.*, 2022, pp. 350–359.
[37] W. Wen et al., "TernGrad: Ternary gradients to reduce communication in distributed deep learning," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 1508–1518.
[38] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "QSGD: Communication-efficient SGD via gradient quantization and encoding," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 1709–1720.
[39] Y. Yu, J. Wu, and L. Huang, "Double quantization for communication-efficient distributed optimization," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 4438–4449.
[40] S. Shi et al., "A distributed synchronous SGD algorithm with global top-k sparsification for low bandwidth networks," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst.*, 2019, pp. 2238–2247.
[41] C.-Y. Chen et al., "ScaleCom: Scalable sparsified gradient compression for communication-efficient distributed training," in *Proc. Adv. Neural Inf. Process. Syst.*, 2020, pp. 13551–13563.
[42] S. Shi et al., "Communication-efficient distributed deep learning with merged gradient sparsification on GPUs," in *Proc. IEEE Conf. Comput. Commun.*, 2020, pp. 406–415.
[43] D. Basu, D. Data, C. Karakus, and S. Diggavi, "Qsparse-local-SGD: Distributed SGD with quantization, sparsification and local computations," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 14695–14706.
[44] F. Sattler, S. Wiedemann, K.-R. Müller, and W. Samek, "Robust and communication-efficient federated learning from non-i.i.d data," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 31, no. 9, pp. 3400–3413, Sep. 2020.
[45] K. Hu, C. Wu, and E. Zhu, "HGC: Hybrid gradient compression in distributed deep learning," in *Proc. Int. Conf. Artif. Intell. Secur.*, 2021, pp. 15–27.
[46] F. Niu, B. Recht, C. Ré, and S. J. Wright, "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent," 2011, *arXiv:1106.5730*.
[47] Q. Ho et al., "More effective distributed ML via a stale synchronous parallel parameter server," in *Proc. Adv. Neural Inf. Process. Syst.*, 2013, pp. 1223–1231.
[48] C. Chen, W. Wang, and B. Li, "Round-robin synchronization: Mitigating communication bottlenecks in parameter servers," in *Proc. IEEE Conf. Comput. Commun.*, 2019, pp. 532–540.
[49] W. Wang, C. Zhang, L. Yang, K. Chen, and K. Tan, "Addressing network bottlenecks with divide-and-shuffle synchronization for distributed dnn training," in *Proc. IEEE Conf. Comput. Commun.*, 2022, pp. 320–329.
[50] M. Cho, U. Finkler, and D. Kung, "BlueConnect: Novel hierarchical all-reduce on multi-tired network for deep learning," in *Proc. 2nd SysML Conf.*, 2019, pp. 241–251.
[51] X. Wan, H. Zhang, H. Wang, S. Hu, J. Zhang, and K. Chen, "Rat-resilient allreduce tree for distributed machine learning," in *Proc. 4th Asia-Pacific Workshop Netw.*, 2020, pp. 52–57.
[52] S. Wang, D. Li, J. Zhang, and W. Lin, "CEFS: Compute-efficient flow scheduling for iterative synchronous applications," in *Proc. 16th Int. Conf. Emerg. Netw. EXperiments Technol.*, 2020, pp. 136–148.
[53] S. Wang, D. Li, and J. Geng, "Geryon: Accelerating distributed CNN training by network-level flow scheduling," in *Proc. IEEE Conf. Comput. Commun.*, 2020, pp. 1678–1687.
[54] L. Luo, M. Liu, J. Nelson, L. Ceze, A. Phanishayee, and A. Krishnamurthy, "Motivating in-network aggregation for distributed deep neural network training," in *Proc. Workshop Approx. Comput. Across Stack*, 2017.
[55] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis, "In-network computation is a dumb idea whose time has come," in *Proc. 16th ACM Workshop Hot Topics Netw.*, 2017, pp. 150–156.
[56] A. Sapio et al., "Scaling distributed machine learning with in-network aggregation," in *Proc. USENIX Conf. Networked Syst. Des. Implementation*, 2021, pp. 785–808.
[57] C. Lao et al., "ATP: In-network aggregation for multi-tenant learning," in *Proc. USENIX Conf. Networked Syst. Des. Implementation*, 2021, pp. 741–761.

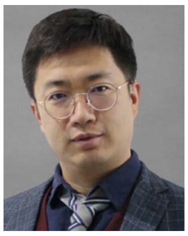
Qingyang Duan received the bachelor's degree from the School of Information Science and Technology, Fudan University, in 2020. He is currently working toward the master's degree with the School of Information Science and Technology, Fudan University. His research interests include computer network systems and distributed machine learning.

Chao Peng received the bachelor's degree from the School of Information Science and Technology, Fudan University, in 2022. He is currently working toward the master's degree with the School of Information Science and Technology, Fudan University. His research interests include computer network systems and distributed machine learning.

Zeqin Wang received the bachelor's degree from the School of Information Science and Technology, Fudan University, in 2021. He is currently working toward the master's degree with the School of Information Science and Technology, Fudan University. His research interests include distributed machine learning and communication network.



Yuedong Xu received the BS degree from Anhui University, the MS degree from the Huazhong University of Science and Technology, and the PhD degree from the Chinese University of Hong Kong. From 2009 to 2012, he held a postdoctoral position with INRIA Sophia Antipolis and Université d'Avignon, France. He is a professor with the School of Information Science and Technology, Fudan University, China. He has published nearly 20 conference and journal papers in premium vents such as CoNEXT, Mobisys, Mobihoc, Infocom and *IEEE/ACM Transactions on Networking*. His research interests include performance evaluation, optimization, machine learning and economic analysis of communication networks and mobile computing.



Shaoteng Liu received the BS and MS degrees in microelectronics from Fudan University, Shanghai, China, and the PhD degree in computer system from the KTH Royal Institute of Technology, Sweden. He served as a technical expert in Huawei. His research interests include electronic systems, networks-on-chip, network coding, optimization, and machine learning. He has published several papers in DATE, NoCS, Infocom, TVLSI and so on.

Jun Wu (Senior Member, IEEE) received the BS degree in information engineering and the MS degree in communication and electronic system from Xidian University, in 1993 and 1996, respectively, and the PhD degree in signal and information processing from the Beijing University of Posts and Telecommunications, in 1999. He is a professor with the School of Computer Science, Fudan University. He was a professor with the Department of Computer Science and Technology, Tongji University. His research interests include wireless network, machine learning, and signal processing.

John C. S. Lui (Fellow, IEEE) received the PhD degree in computer science from the University of California at Los Angeles, CA, USA, in 1992. He is currently the Choh-Ming Li chair professor with the Department of Computer Science and Engineering, Chinese University of Hong Kong, Hong Kong. His research interests include machine learning, online learning (e.g., multiarmed bandit, reinforcement learning), network science, future internet architectures and protocols, network economics, network/system security, and large scale storage systems. He was the recipient of Departmental Teaching awards and the CUHK Vice-Chancellor's Exemplary Teaching Award. He was also the co-recipient of the Best Paper Award in the IFIP WG 7.3 Performance 2005, IEEE/IFIP NOMS 2006, SIMPLEX 2013, and ACM RecSys 2017. From 2011 to 2015, he was the past chair of the ACM SIGMETRICS. He is an elected member of the IFIP WG 7.3, fellow of ACM, senior research fellow of the Croucher Foundation.