US-Byte: An Efficient Communication Framework for Scheduling Unequal-sized Tensor Blocks in Distributed Deep Learning

Yunqi Gao, *Student Member, IEEE,* Bing Hu, *Senior Member, IEEE,*
Mahdi Boloursaz Mashhadi, *Senior Member, IEEE,* A-Long Jin, *Student Member, IEEE,*
Pei Xiao, *Senior Member, IEEE* and Chunming Wu, *Senior Member, IEEE*

Abstract—The communication bottleneck severely constrains the scalability of distributed deep learning, and efficient communication scheduling accelerates distributed DNN training by overlapping computation and communication tasks. However, existing approaches based on tensor partitioning are not efficient and suffer from two challenges: (1) the fixed number of tensor blocks transferred in parallel can not necessarily minimize the communication overheads; (2) although the scheduling order that preferentially transmits tensor blocks close to the input layer can start forward propagation in the next iteration earlier, the shortest per-iteration time is not obtained. In this paper, we propose an efficient communication framework called US-Byte. It can schedule unequal-sized tensor blocks in a near-optimal order to minimize the training time. We build the mathematical model of US-Byte by two phases: (1) the overlap of gradient communication and backward propagation, and (2) the overlap of gradient communication and forward propagation. We theoretically derive the optimal solution for the second phase and efficiently solve the first phase with a low-complexity algorithm. We implement the US-Byte architecture on PyTorch framework. Extensive experiments on two different 8-node GPU clusters demonstrate that US-Byte can achieve up to 1.26x and 1.56x speedup compared to ByteScheduler and WFBP, respectively. We further exploit simulations of 128 GPUs to verify the potential scaling performance of US-Byte. Simulation results show that US-Byte can achieve up to 1.69x speedup compared to the state-of-the-art communication framework.

Index Terms—Distributed Deep Learning, Data Parallelism, Communication Scheduling, Tensor Partitioning, Tensor Fusion.

1 Introduction

As the scale of DNNs grows and the amount of data explodes, DNN training becomes increasingly time-consuming. Data parallelism is widely used to accelerate DNN training in distributed deep learning [1], [2], [3], [4]. Data parallelism consists of three main components: forward propagation, backward propagation, and gradient aggregation. In traditional data parallelism with synchronous stochastic gradient descent (SGD), multiple nodes simultaneously train a DNN, and the gradients are aggregated to update the model parameters in each iteration [5], [6]. There are various factors that limit the scaling efficiency of data parallelism, such as energy consumption [7], [8], dataset privacy [9], [10], [11], and imbalance of computing resources [12]. In addition, with large-scale datasets, models, and clusters, intensive data communication due to gradient aggregation among nodes introduces a huge additional time overhead for data parallelism and limits its scalability [4], [5], [13]. Consequently, communication performance has become a significant bottleneck in distributed DNN training.

The DNN training process can generally be represented as a dependent directed acyclic graph (DAG) in mainstream deep learning engines, such as PyTorch [14], TensorFlow [15], and MXNet [16]. In the all-reduce architecture, one iteration of a worker mainly consists of two sets of layer-wise computation operations (forward propagation and backward propagation) and a set of communication operations (gradient aggregation). Meanwhile, the execution order of these operations is determined by the underlying DAG. Since these two types of operations do not occupy the same resources, researchers find that communication can be overlapped with computation tasks in data parallelism to reduce the time of one iteration without affecting the convergence performance of the DNN [13]. Wait-free backward propagation (WFBP) has been adopted in most deep learning frameworks [17]. WFBP transmits the gradient tensor of each layer once it completes backward propagation.

Most recently, communication scheduling has been studied to improve the communication performance of distributed DNN training [4], [5], [18], [19], [20], [21]. The main idea of communication scheduling is to change the execution order of communication operations and better overlap computation and communication tasks. A practical method is tensor fusion [18], [22], which merges tensors of multiple layers into one large tensor for transmission to reduce the communication overheads. Notably, the commu-

- *Yunqi Gao Bing Hu and Chunming Wu are with Zhejiang University, Hangzhou 310027, China. E-mail: gaoyunqi1999, binghu, wuchunming@zju.edu.cn.*
- *Mahdi Boloursaz Mashhadi and Pei Xiao are with 5GIC & 6GIC, Institute for Communication Systems (ICS), University of Surrey, United Kingdom. E-mail: m.boloursazmashhadi, p.xiao@surrey.ac.uk.*
- *A-Long Jin is with the University of Hong Kong, Hong Kong, China. E-mail: ajin@eee.hku.hk.*

This work was supported in part by the National Key Research and Development Project under Grant 2022YFB2901600, in part by the General Program of National Natural Science Foundation of China under Grant 61971377, and in part by the Key Project of Natural Science Foundation of Zhejiang Province under Grant LZ22F010008.
(Corresponding author: Bing Hu.)

nication overhead is mainly caused by the synchronization between nodes in the cluster at the beginning of gradient aggregation (e.g., an all-reduce operation) [5]. It is related to software and hardware latency, where software latency is the latency of the communication libraries and protocols used during synchronization, and hardware latency comes from the latency of the network bandwidth, topology, and devices during synchronization. Another popular method is to prioritize tensor transmission for different layers. ByteScheduler [19], the state-of-the-art communication scheduler, proposes tensor partitioning and priority scheduling. In ByteScheduler, each layer's tensor is partitioned into multiple small tensor blocks according to a fixed threshold (partition size), and small tensor blocks close to the input layer are transmitted preferentially. This is because forward propagation of the next iteration can be started earlier if the tensor close to the input layer is aggregated first.

However, there exist two challenges in the ByteScheduler. Firstly, when using the all-reduce architecture, the transmission of each small tensor block requires an all-reduce operation. Then, the communication overheads of enabling the all-reduce operations results in low bandwidth utilization when transmitting many small tensor blocks. ByteScheduler hides the communication overheads by transmitting tensor blocks in parallel (Sec. 2.3). Though performant, the fixed number of tensor blocks being transmitted in parallel does not necessarily result in the least communication overheads. Secondly, when transmitting tensors, tensor blocks close to the input layer have higher priority. Although this scheduling order can start forward propagation of the next iteration earlier, it does not necessarily obtain the shortest per-iteration time. To address the above two challenges, we argue that merging small tensor blocks into one large tensor block for transmission based on tensor fusion (i.e., changing the partition size of tensor blocks) can significantly reduce the unnecessary communication overhead. Meanwhile, we also find that when transmitting tensor blocks with different partition sizes, the communication overhead can be further reduced by adjusting the scheduling order of each tensor block.

In this paper, we propose an efficient communication framework called US-Byte. It can partition the tensor of each layer using variable partition sizes and transmit the tensor blocks with the near-optimal scheduling order. Specifically, US-Byte first partitions each tensor into small tensor blocks based on tensor partitioning and priority scheduling. Then, US-Byte dynamically determines the number of neighboring small tensor blocks from the same layer being merged and the scheduling order of the merged tensor blocks to minimize the time of one iteration. We build the mathematical model of US-Byte by two phases: (1) the first phase is the overlap between gradient communication and backward propagation, and (2) the second phase is the overlap between gradient communication and forward computation. Notably, we prove that sequential scheduling is optimal in the second phase without considering tensor fusion across layers. We efficiently solve the first phase with an algorithm of complexity $O\left(L^3\right)$, where L is the number of layers in the DNN. Additionally, we implement the US-Byte architecture on PyTorch frameworks and make it open-source [1]. To validate the effectiveness of our proposed US-Byte, we evaluate its performance using six modern DNNs on two GPU clusters with 10Gbps Ethernet and 100Gbps bandwidth interconnects. On the Nvidia Tesla V100 GPU clusters with 10GbE, US-Byte outperforms ByteScheduler by 11%-26% and WFBP by 21%-56% in terms of training speed. On the Nvidia RTX 3090 GPU clusters, US-Byte outperforms ByteScheduler by up to 24% when the bandwidth level is limited to 20Gbps. To explore its scaling efficiency on large-scale clusters, we conduct simulations (due to limited hardware resources) on a 128-node cluster. The results in the simulation cluster show that US-Byte outperforms ByteScheduler by 28%-69% and WFBP by 234%-332%.

The contributions of this paper are summarized as follows:

- We propose a communication framework (US-Byte) that can schedule unequal-sized tensor blocks in a near-optimal order to minimize the distributed training time of DNNs.
- We model US-Byte mathematically and efficiently find its near-optimal solution.
- We implement US-Byte architecture on PyTorch frameworks and make it open-source.
- We conduct extensive experiments on two GPU clusters to evaluate US-Byte performance. Experimental results demonstrate that US-Byte significantly accelerates the training of DNNs compared to state-of-the-art communication frameworks.

The rest of the paper is organized as follows. We introduce the background and motivation in Section 2. We build the mathematical model of US-Byte in Section 3. We theoretically analyze the optimal solution of the second phase of US-Byte and present an efficient algorithm to obtain the scheduling order of the first phase in Section 4. The system implementation of US-Byte is elaborated in Section 5. Section 6 evaluates the performance of US-Byte compared to existing communication frameworks. We describe the related work and make a discussion in Section 7. Finally, we conclude Section 8.

2 Background and Motivation

2.1 Distributed Deep Learning

DNN training is an iterative process of minimizing a loss function over a large dataset. The iteration is repeated until the prediction accuracy of the DNN converges to the expected value.

Forward propagation and backward propagation. In each iteration, the large dataset is partitioned into multiple mini-batches. A mini-batch of data travels from the first layer to the last layer of the DNN and generates a loss. This process is called forward propagation. Then, the gradient of the loss function with respect to the DNN parameters is calculated from the last layer to the first layer. This process is called backward propagation. Finally, the DNN parameters are updated by stochastic gradient descent using the calculated gradient. After that, a new mini-batch of data is sent to the DNN, and the above process is repeated iteratively.

1. https://github.com/ZJU-CNLAB/US-Byte

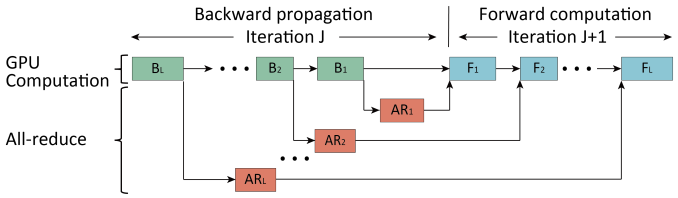


Fig. 1: Computation-communication dependency DAG in distributed DNN training.

Data parallelism. Since training large-scale DNNs is a time-consuming task on a single node, distributed training via data parallelism has become a popular practice [3]. Data parallelism is one of the most prevalent methods to accelerate distributed DNN training. In data parallelism, each node contains a complete replica of the DNN and is assigned a mini-batch of data in each iteration. Then, multiple nodes perform DNN training simultaneously, and the gradients are aggregated to update the parameters [5], [6].

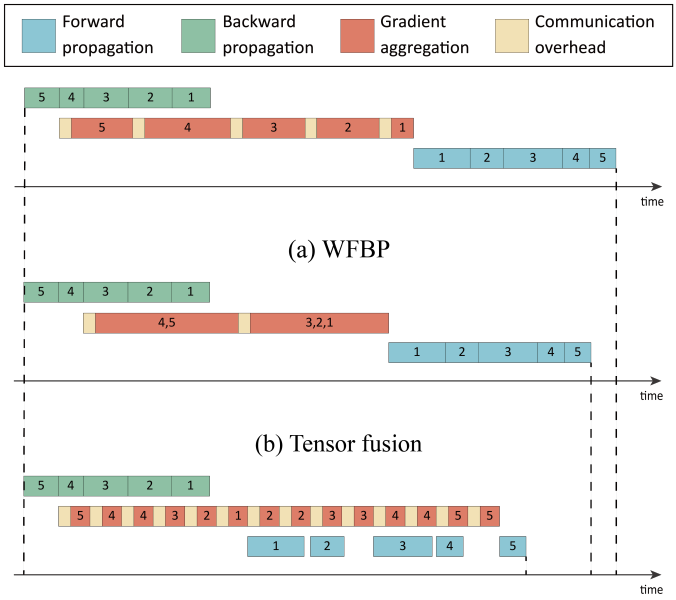
All-reduce architecture There are two common communication architectures used for gradient aggregation in each iteration of data parallelism, including Parameter Server [23] and All-reduce [22]. In this paper, we focus on the all-reduce architecture. All-reduce is a centerless collective operation that does not require central servers to update the parameters. Each worker collects the others' gradients and disperses the averaged gradients to the others. There are various algorithms to implement the all-reduce operation, the most popular of which is ring-allreduce. Each tensor (i.e., the gradient aggregated by the all-reduce operation) is evenly chunked according to the number of workers, and the gradient aggregation is completed when all the tensor chunks have traveled through the ring. Ring-allreduce is widely used in deep learning frameworks such as Horovod [22] and PaddlePaddle [2], and it has also been shown to be bandwidth-optimal [24].

2.2 Communication Scheduling

Dependency directed acyclic graph (DAG). The layer-wise structure of the DNN enables it to be represented as a dataflow graph in one iteration. The dataflow graph is usually a directed acyclic graph (DAG). Modern deep learning framework engines execute the DAG to train DNNs. Every operation in the DAG is executed immediately once its dependent operations are completed. Fig. 1 shows an example of a dependency DAG under the All-reduce architecture. We define F_l, B_l, and AR_l as forward propagation, backward propagation, and all-reduce communication (gradient aggregation) of layer l. F_l depends on F_{l-1} and AR_l, B_l depends on B_{l+1}, and AR_l depends on B_l [19].

Communication scheduling is commonly used to accelerate distributed deep learning, which reduces the time of an iteration by overlapping communication and computation. Wait-free backward propagation (WFBP) has been adopted in modern deep learning frameworks such as PyTorch, TensorFlow, and MXNet. From Fig. 2a, WFBP utilizes the layer-wise structure of DNN to overlap gradient aggregation and backward propagation. In WFBP, AR_l is executed immediately once B_l and AR_{l-1} are completed.

2. http://en.paddlepaddle.org/en



(c) Tensor partitioning + Priority scheduling

Fig. 2: An example of training a 5-layer network showing one iteration time under three communication scheduling framework.

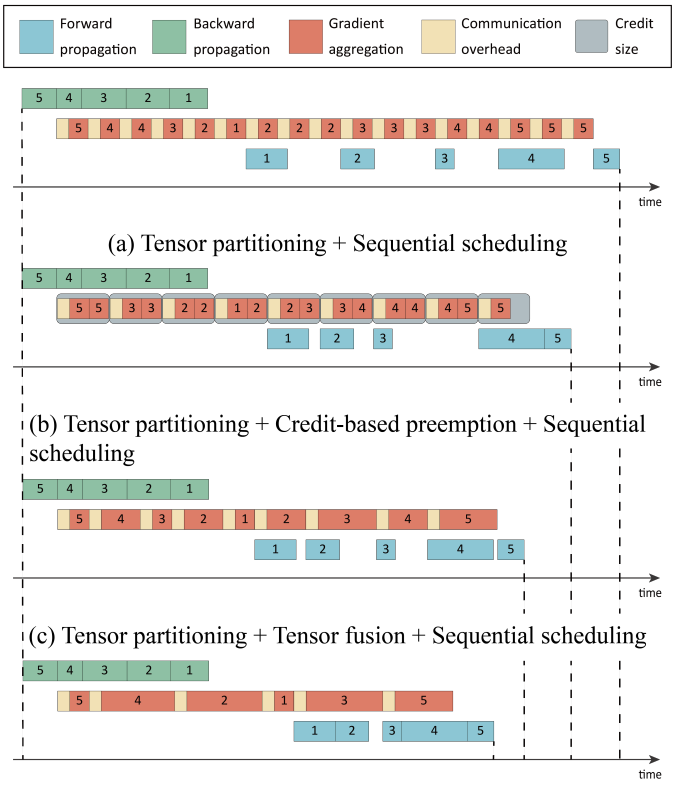
Currently, tensor fusion and tensor partitioning are two prevalent communication scheduling strategies.

Tensor fusion. Since enabling an all-reduce operation will introduce a portion of communication overhead, the separate transmission of tensors (gradients) for each layer in WFBP degrades bandwidth utilization. Therefore, tensor fusion effectively reduces communication overhead by merging tensors of multiple layers into one large tensor for transmission. In the example from Fig. 2b, layer 4 and layer 5 are merged for transmission while layer 1, layer 2 and layer 3 are merged for transmission. Compared to WFBP, tensor fusion reduces the time of one iteration.

Tensor partitioning and priority scheduling. The backward propagation of the DNN training determines that the tensor of the layers close to the output layer is computed earlier. However, the order of forward propagation is from the input layer to the output layer. According to the dependency DAG of the DNN in Fig. 1, F_l will be executed earlier if AR_l is completed earlier. Therefore, ByteScheduler proposes tensor partitioning and priority scheduling, which effectively overlap gradient aggregation and forward propagation. From Fig. 2c, tensor partitioning slices the tensor of each layer into multiple tensor blocks according to a fixed threshold (partition size). Then, the priority scheduling makes the tensor blocks of the layers close to the input layer be transmitted first to ensure that the forward propagation of the next iteration starts earlier. Meanwhile, we call this scheduling order sequential scheduling.

2.3 Motivation

From the example in Fig. 2, although tensor partitioning and priority scheduling achieve the shortest iteration time, we observe that the total gradient aggregation time (including



(d) Tensor partitioning + Tensor fusion + Optimal scheduling

Fig. 3: An example of training a 5-layer network showing performance gain using tensor fusion and optimal scheduling order under tensor partitioning.

communication overhead) in Fig. 2c is longer than the total gradient aggregation time in Fig. 2a and Fig. 2b. This is because the transmission of each tensor block needs to wait for a portion of communication overhead, which results in lower bandwidth utilization.

To reduce the communication overhead of transmitting tensor blocks, ByteScheduler uses credit-based preemption, which works like a sliding window mechanism. Credit-based preemption allows multiple tensor blocks to be transferred in parallel by setting the credit size, and an example with credit size = 2 is shown in Fig. 3b. However, credit-based preemption breaks tensor priority in sequential scheduling. Although credit-based preemption reduces communication overhead, it cannot guarantee that tensor blocks with higher priority will be sent promptly. For example, in Fig. 3b, due to the preemption of tensor blocks of layer 5, tensor blocks of layer 4 cannot be transmitted in time after being computed. Therefore, credit-based preemption is only a suboptimal solution to reduce communication overhead.

We find that neighboring tensor blocks from the same layer can be merged for transmission to reduce unnecessary communication overhead. Comparing Figs. 3a and 3c, changing the partition size of the tensor block through tensor fusion can shorten the total gradient aggregation time without affecting sequential scheduling. Moreover, sequential scheduling is not optimal when the tensor blocks being transmitted each time are not equal in partition size. Adjusting the scheduling order of unequal-sized tensor blocks can further reduce the communication overhead compared to sequential scheduling. For example, in Fig. 3d, raising the priority of tensor blocks of layer 4 above that of layer 3 can shorten the iteration time compared to Fig. 3c.

Consequently, we optimize and adapt the partition size and scheduling order of tensor blocks to effectively reduce the communication overhead and obtain shorter time of one iteration based on tensor partitioning.

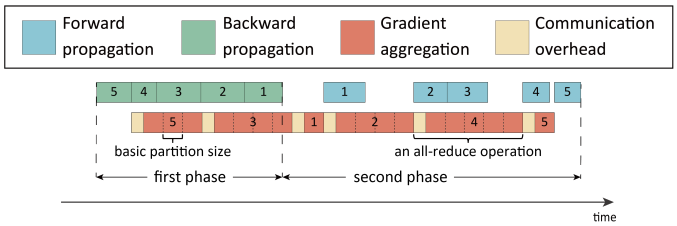


Fig. 4: An example to describe the mathematical model of US-Byte.

TABLE 1: Summary of notations used in US-Byte

Name	Description
N	The number of nodes in the cluster.
a	Communication overhead of an all-reduce operation.
b	Transmission time per byte of all-reduce.
δ_1	Number of all-reduce operations in the first phase.
δ_2	Number of all-reduce operations in the second phase.
S_p	Basic partition size.
$O_{num}^{(i)}$	Number of basic tensor blocks transmitted in the i-th all-reduce operation, where $1 \leq i \leq (\delta_1 + \delta_2)$.
$O_{lay}^{(i)}$	Serial number of layer that the tensor block transmitted in i-th all-reduce operation belongs to.
L	The number of network layers.
$M^{(l)}$	The tensor size of layer l that needs to be transmitted in each iteration.
$m^{(l)}$	Number of basic tensor blocks of layer l in each iteration.
$m_{re}^{(l)}$	Number of basic tensor blocks remaining in layer l that have been computed but not transmitted.
$\tau_b^{(l)}$	The timestamp when layer l begins the backward propagation.
$t_b^{(l)}$	Time of the backward propagation of layer l in each iteration.
$\tau_{ar}^{(i)}$	The timestamp when the i-th all-reduce operation begins.
$\tau_f^{(l)}$	The timestamp when layer l begins the forward propagation.
$t_f^{(l)}$	Time of the forward propagation of layer l in each iteration.
Δt	Time error due to the fact that the size of the last basic tensor block from the same layer may be smaller than S_p.
$P_1^{(l)}$	The priority of the tensor of layer l in the first phase. The smaller the P_1 value ($1 \leq P_1^{(l)} \leq L$), the higher the priority.
$P_2^{(l)}$	The priority of the tensor of layer l in the second phase. The smaller the P_2 value ($1 \leq P_2^{(l)} \leq L$), the higher the priority.
t_{iter}	Time of one iteration.

3 Mathematical Model of US-Byte

In this section, we build the mathematical model of US-Byte. Table 1 provides all notations used in US-Byte.

As shown in Fig. 4, we divide US-Byte into two phases to model. These two phases are separated by the timestamp when layer 1 ends the backward propagation. The

first phase represents the overlapping process of gradient aggregation and backward propagation. The second phase describes the overlapping process of gradient aggregation and forward propagation. The priorities of the tensor of layer l are defined as $P_1^{(l)}$ and $P_2^{(l)}$ in the first and second phases, respectively, which determine the scheduling order of the tensor. We define the partition size (the fixed threshold) from tensor partitioning and sequential scheduling as the basic partition size, denoted by S_p, and the tensor of each layer is partitioned into multiple basic tensor blocks according to the basic partition size. In the US-Byte's model, each tensor block is obtained by merging multiple basic tensor blocks. Each all-reduce operation transmits one tensor block. We denote an all-reduce operation by a two-tuple $< O_{num}^{(i)}, O_{lay}^{(i)} >$, which means that in the i-th all-reduce operation, the tensor block of the layer $O_{lay}^{(i)}$ is transmitted and the tensor block consists of $O_{num}^{(i)}$ basic tensor blocks.

We first model the relationship between the time of an all-reduce operation and the tensor block size (consisting of multiple basic partition sizes) as follows:

$$T_{ar}(O_{num}^{(i)}) = \begin{cases} 0 & O_{num}^{(i)} = 0 \\ a + b \times S_p \times O_{num}^{(i)} & O_{num}^{(i)} > 0 \end{cases}, \quad (1)$$

where a and b are two constants that are related to the hardware cluster and are independent of $O_{num}^{(i)}$. Then, we formulate the mathematical model of US-Byte as follows:

The first phase. For simplicity, we define the time of one iteration equal to the time difference between the timestamp when layer L begins the backward propagation in iteration J and the timestamp when layer L ends the forward computation in iteration $(J+1)$ (see Fig. 1). Meanwhile, we assume that the start timestamp of backward propagation is 0, i.e., $\tau_b^{(L)} = 0$. Then, in the first phase, the timestamp when each layer begins the backward propagation, denoted by $\tau_b^{(l)}$, can be represented by

$$\tau_b^{(l)} = \begin{cases} 0 & l = L \\ \tau_b^{(l+1)} + t_b^{(l+1)} & l < L \end{cases}, \quad (2)$$

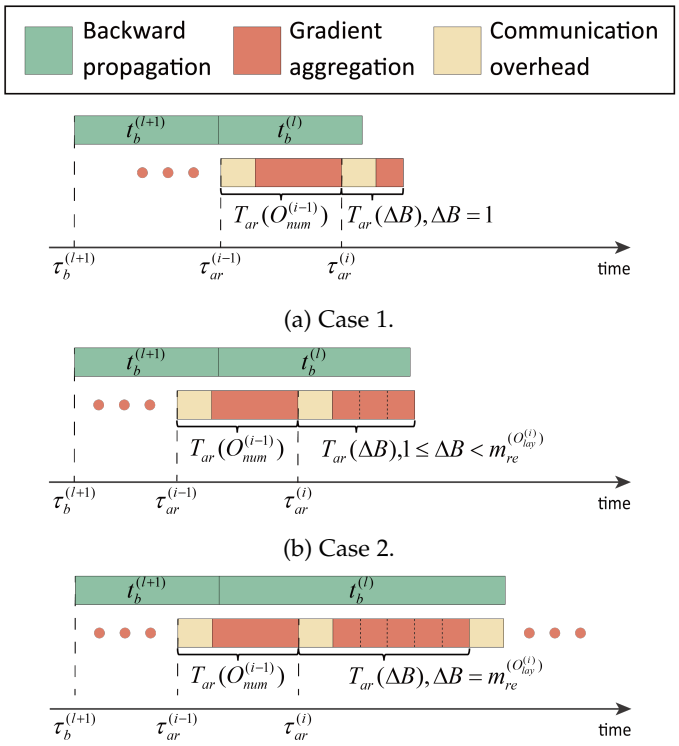
where $t_b^{(l)}$ represents the time of the backward propagation of layer l in each iteration.

The i-th all-reduce operation in the first phase can start if the following two conditions are satisfied: (1) the tensor of layer $O_{lay}^{(i)}$ has been calculated; (2) the $(i-1)$-th all-reduce operation has finished. Therefore, the timestamp when the i-th all-reduce operation $(1 \le i \le \delta_1)$ begins, denoted by $\tau_{ar}^{(i)}$, can be represented by

$$\tau_{ar}^{(i)} = \begin{cases} \tau_b^{\left(O_{lay}^{(i)}\right)} + t_b^{\left(O_{lay}^{(i)}\right)} & i = 1 \\ max\left\{\tau_b^{\left(O_{lay}^{(i)}\right)} + t_b^{\left(O_{lay}^{(i)}\right)}, \tau_{ar}^{(i-1)} + T_{ar}(O_{num}^{(i-1)})\right\} & i > 1 \end{cases}, \quad (3)$$

where it is obvious that $O_{lay}^{(1)} = L$.

As backward propagation proceeds, the two-tuples of the all-reduce operations in the first phase can be calculated according to the priority of tensor of each layer. Assume that the i-th all-reduce operation will start during the backward propagation of layer l, i.e., the tensor of layer $O_{lay}^{(i)}$ has the highest priority at this point. From Fig. 5, the number of basic tensor blocks that need to be merged for transmission before layer l ends backward propagation can be calculated in three cases.



(a) Case 1.

(b) Case 2.

(c) Case 3.

Fig. 5: Three cases when executing the i-th all-reduce operation during the backward propagation of layer l.

Case 1. *From Fig. 5a, the time difference between the end timestamp of the backward propagation of layer l and the end timestamp of the $(i-1)$th all-reduce operation is less than the communication overhead a.*

$$\left(\tau_b^{(l)} + t_b^{(l)}\right) - \tau_{ar}^{(i)} < a. \quad (4)$$

When Case 1 holds, the number of basic tensor blocks that need to be merged in the ith all-reduce operation is $\Delta B = 1$.

Case 2. *From Fig. 5b, the time difference between the end timestamp of the backward propagation of layer l and the end timestamp of the $(i-1)$th all-reduce operation is insufficient to transmit all remaining tensor blocks of layer $O_{lay}^{(i)}$.*

$$a \le \left(\tau_b^{(l)} + t_b^{(l)}\right) - \tau_{ar}^{(i)} < T_{ar}(m_{re}^{\left(O_{lay}^{(i)}\right)}), \quad (5)$$

where $m_{re}^{(l)}$ represents the number of basic tensor blocks remaining in layer l that have been computed but not transmitted. When Case 2 holds, the number of basic tensor blocks that need to be merged in the ith all-reduce operation is ΔB, where ΔB can be calculated by

$$\Delta B = \left\lceil \frac{\left(\tau_b^{(l)} + t_b^{(l)}\right) - \tau_{ar}^{(i)} - a}{b \times S_p} \right\rceil. \quad (6)$$

Algorithm 1 Get $< O_{num}^{(i)}, O_{lay}^{(i)} >$ in the first phase

Input: a, b, S_p, L, $M[1...L]$, $P_1[1...L]$, $t_b[1...L]$
Output: $\delta_1, O_{num}[1...\delta_1], O_{lay}[1...\delta_1], \tau_{ar}[1...\delta_1]$
1: **for** $l = 1 \rightarrow L$ **do**
2: $m[l] = \lceil M[l]/S_p \rceil$; $m_{re}[l] = 0$;
3: $\tau_b[L] = 0$;
4: **for** $l = L - 1 \rightarrow 1$ **do**
5: $\tau_b[l] = \tau_b[l+1] + t_b[l+1]$;
6: Initialize $\tau_{ar}[1] = \tau_b[L] + t_b[L]$, $i = 1$, $L_{old} = 0$;
7: **for** $l = L \rightarrow 1$ **do**
8: **while** $(\tau_b[l] + t_b[l]) - \tau_{ar}[i] > 0$ **do**
9: $O_{lay}[i]$=CALCULATE_$O_{lay}(L, P_1, m_{re})$;
10: **if** $O_{lay}[i] == 0$ **then**
11: $\tau_{ar}[i] = \tau_b[l] + t_b[l]$;
12: break;
13: **else if** $O_{lay}[i] == L_{old}$ **then**
14: $i = i - 1$;
15: **if** Case 1 **then**
16: $\Delta B = 1$;
17: **else if** Case 2 **then**
18: $\Delta B = \left\lfloor \frac{(\tau_b[l] + t_b[l]) - \tau_{ar}[i] - a}{b \times S_p} \right\rfloor$;
19: **else if** Case 3 **then**
20: $\Delta B = m_{re}[O_{lay}[i]]$;
21: $O_{num}[i] = \Delta B$;
22: $m_{re}[O_{lay}[i]] = m_{re}[O_{lay}[i]] - O_{num}[i]$;
23: Δt=CALCULATE_$\Delta t(b, S_p, M, m, m_{re}, O_{lay}, i)$;
24: $\tau_{ar}[i+1] = \tau_{ar}[i] + T_{ar}(O_{num}[i]) - \Delta t$;
25: $L_{old} = O_{lay}[i]$;
26: $i = i + 1$;
27: $m_{re}[l] = m[l]$;
28: $\delta_1 = i - 1$;
29: **return** $\delta_1, O_{num}[1...\delta_1], O_{lay}[1...\delta_1], \tau_{ar}[1...\delta_1], m_{re}[1...L]$;
30: **procedure** CALCULATE_$O_{lay}(L, P_1, m_{re})$
31: $layer = 0$; $P = L + 1$;
32: **for** $l = L \rightarrow 1$ **do**
33: **if** $m_{re}[l] > 0$ and $P_1[l] < P$ **then**
34: $layer = l$; $P = P_1[l]$;
35: **return** $layer$;
36: **procedure** CALCULATE_$\Delta t(b, S_p, M, m, m_{re}, O_{lay}, i)$
37: **if** $m_{re}[O_{lay}[i]] == 0$ **then**
38: $\Delta t = (m[O_{lay}[i]] - M[O_{lay}[i]]/S_p) \times b \times S_p$;
39: **else**
40: $\Delta t = 0$;
41: **return** Δt;

Case 3. *From Fig. 5c, the time difference between the end timestamp of the backward propagation of layer l and the end timestamp of the $(i-1)$th all-reduce operation can transmit all remaining tensor blocks of layer $O_{lay}^{(i)}$.*

$$\left(\tau_b^{(l)} + t_b^{(l)}\right) - \tau_{ar}^{(i)} \geq T_{ar}(m_{re}^{\left(O_{lay}^{(i)}\right)}). \tag{7}$$

When case 3 holds, the number of basic tensor blocks that need to be merged in the ith all-reduce operation is $\Delta B = m_{re}^{\left(O_{lay}^{(i)}\right)}$. Meanwhile, $O_{num}^{(i)} = m_{re}^{\left(O_{lay}^{(i)}\right)}$.

However, the tensor of each layer is not always an integer multiple of S_p, and the time error due to incomplete basic tensor blocks (size $< S_p$) from layer l, denoted by Δt, can be calculated by:

$$\Delta t = \left(m^{(l)} - M^{(l)}/S_p\right) \times b \times S_p. \tag{8}$$

where $m^{(l)}$ represents the number of basic tensor blocks of layer l in each iteration, and $M^{(l)}$ represents the tensor size of layer l that needs to be transmitted in each iteration.

Since new basic tensor blocks are continuously generated as the backward propagation goes, the basic tensor blocks with the highest priority are constantly changing.

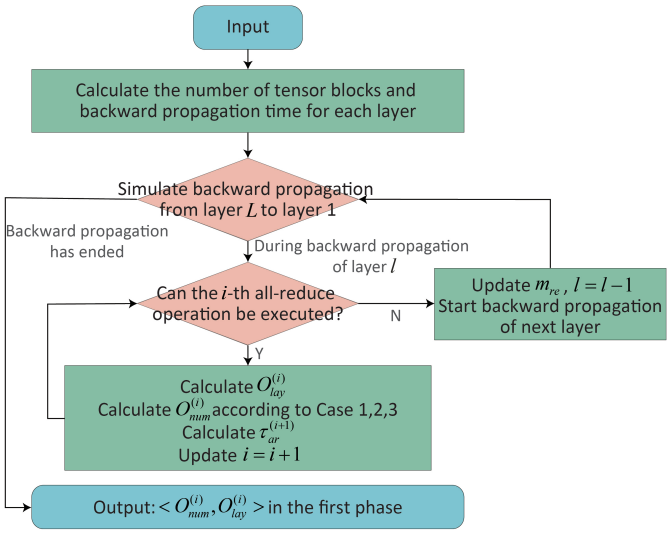


Fig. 6: The flowchart of Algorithm 1.

This makes it difficult to express the relationship between the two-tuple of the all-reduce operation and tensor priority by a fixed function. Thus, we introduce Algorithm 1 to calculate $< O_{num}^{(i)}, O_{lay}^{(i)} >$ in the first phase according to $P_1^{(l)}$. Algorithm 1 (line 1-2) first calculates the number of basis tensor blocks of each layer and initializes the number m_{re} of basic tensor blocks remaining in each layer. Line 3-5 obtains the layer-wise start timestamp of the backward propagation according to Eq. 2. Then, line 6-28 calculates the two-tuple of each all-reduce operation in the first phase. Line 27 updates m_{re} to simulate backward propagation. When the i-th all-reduce operation can be executed during the backward propagation of layer l (line 8), line 9 calculates $O_{lay}^{(i)}$ by finding basic tensor block with the highest priority from all remaining basis tensor blocks, line 15-21 obtains $O_{num}^{(i)}$ according to Eqs. 4, 5, 6 and 7, and line 22 updates m_{re}. Three points are worth noting: (1) if all basic tensor blocks have been transmitted, the i-th all-reduce operation can only start when the backward propagation of layer l ends (line 11-13); (2) if two consecutive all-reduce operations transmit tensor blocks from the same layer, the parameter i will not need to be updated (line 13-14); (3) the time error Δt is calculated on line 23 according to Eq. 8 and used to derive the start timestamp of each all-reduce operation (line 24). Meanwhile, to help understanding, the flowchart of Algorithm 1 is shown in Fig. 6.

The second phase. The i-th all-reduce operation in the second phase can start as soon as the $(i-1)$-th all-reduce operation is completed, i.e. the timestamp when the i-th all-reduce operation ($\delta_1 + 1 \leq i \leq \delta_2$) begins can be calculated by

$$\tau_{ar}^{(i)} = \begin{cases} max\left\{\tau_b^{(1)} + t_b^{(1)}, \tau_{ar}^{(\delta_1)} + T_{ar}(O_{num}^{(\delta_1)})\right\} & i = \delta_1 + 1 \\ \tau_{ar}^{(i-1)} + T_{ar}(O_{num}^{(i-1)}) & i > \delta_1 + 1 \end{cases}. \tag{9}$$

Since all remaining tensor blocks will be transmitted in the second phase, it is obvious that $\delta_2 = L$. Thus, we can obtain the relationship between the i-th all-reduce operation

and the tensor priority in the second phase by

$$< O_{num}^{(\delta_1+P_2^{(l)})}, O_{lay}^{(\delta_1+P_2^{(l)})} >=< m_{re}^{(l)}, l >, \quad (10)$$

i.e., the remaining basic tensor blocks of layer l will be transmitted through the $(\delta_1 + P_2^{(l)})$-th all-reduce operation.

The forward propagation of layer l can start if the following two conditions are satisfied: (1) the forward propagation of layer $(l-1)$ has finished; (2) all tensor blocks of layer l have been transmitted. Then, the timestamp when layer l begins the forward propagation, denoted by $\tau_f^{(l)}$, can be calculated by

$$\tau_f^{(l)}=\begin{cases} \tau_{ar}^{(\delta_1+P_2^{(l)})}+T_{ar}(m_{re}^{(l)})-\Delta t & l=1 \\ \max\{\tau_f^{(l-1)}+t_f^{(l-1)}, \tau_{ar}^{(\delta_1+P_2^{(l)})}+T_{ar}(m_{re}^{(l)})-\Delta t\} & l>1 \end{cases}, \quad (11)$$

where the time error Δt can be calculated in the same way as in line 36-41 of Algorithm 1.

Finally, we would like to minimize the time of one iteration, i.e.,

$$\textbf{minimize: } t_{iter} = \tau_f^{(L)} + t_f^{(L)} - \tau_b^{(L)}, \quad (12)$$

where $t_f^{(L)}$ represents the time of the forward propagation of layer l in each iteration.

Among the parameters of the mathematical model of US-Byte, $t_b^{(l)}$, $t_f^{(l)}$, a and b need to be actually measured. $t_b^{(l)}$ and $t_f^{(l)}$ are obtained by averaging multiple iterations before training. The measurement of a and b is described in Section 6.2. In addition, the degree to which US-Byte's model accurately mirrors real-world system timings is also evaluated in Section 6.2.

4 Solution

In this section, we obtain the optimal scheduling order of the second phase of US-Byte by theoretical analysis and propose an efficient algorithm to find the near-optimal scheduling order of the first phase.

4.1 Theoretical Analysis

Theorem 1. *Sequential scheduling is the optimal scheduling order for the second phase of US-Byte, i.e., $P_2^{(l)} = l$.*

Proof. Theorem 1's proof can be found in Appendix. ■

According to Theorem 1, US-Byte is proven to be optimal when overlapping gradient communication and forward computation without tensor fusion across layers. However, if either of the above two conditions is not satisfied, finding the optimal scheduling order for tensor blocks will be an NP-hard problem and cannot be solved in a reasonable time.

4.2 Algorithm

For a given DNN and a hardware cluster, the performance gain of US-Byte is determined by P_1 and P_2. In the mathematical model of US-Byte, both P_1 and P_2 have $L!$ values. Therefore, the complexity of finding the optimal scheduling order for US-Byte is $O\left(L!^2\right)$. Then, according to Theorem 1, we simplify the complexity to $O\left(L!\right)$ without loss of optimality.

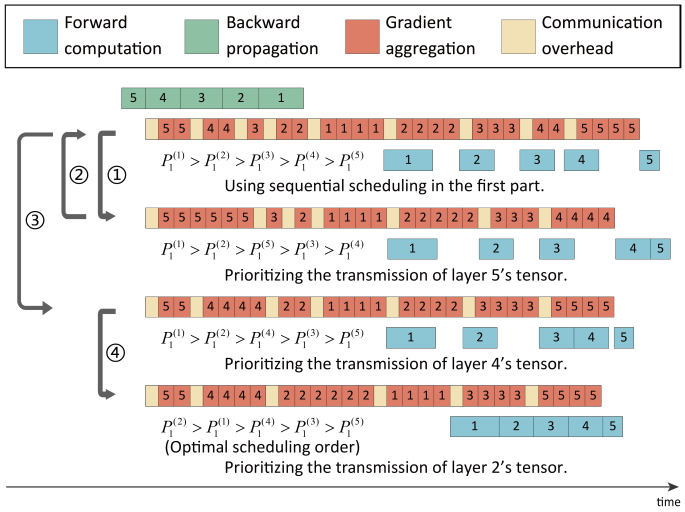


Fig. 7: An example of the proposed greedy algorithm shows the process of finding the near-optimal scheduling order of the first phase. The black arrows with serial numbers represent the execution process of the algorithm.

However, finding the optimal scheduling order for the first phase is still an NP-hard problem. To this end, we propose a greedy algorithm with low complexity to obtain an approximate optimal solution. Fig. 7 illustrates the process of this greedy algorithm with an example. We first initialize the priority of the tensor of each layer based on sequential scheduling. Then, we sequentially raise the priority of the tensor of layer l from layer L to layer 2 by judging whether prioritizing the transmission of the tensor of layer l can shorten the time of one iteration. For example, from Fig. 7, prioritizing the transmission of layer 5's tensor leads to longer iteration time, however, prioritizing the transmission of tensors of layer 4 and layer 2 can achieve a better performance gain.

When prioritizing the transmission of the tensor of layer l, if the basic tensor blocks of layer l is first transmitted by the i-th all-reduce operation in the first phase, the relationship between the timestamp when all tensor blocks of layer l have been transmitted in the first phase and the start timestamp of the backward propagation of each layer can be represented by

$$\tau_b^{(k)} + t_b^{(k)} < \tau_{ar}^{(i)} + T_{ar}(m^{(l)}) - \Delta t \leq \tau_b^{(k-1)} + t_b^{(k-1)}. \quad (13)$$

According to Eq. 13, the priority of the tensor of layer l should be raised above that of layer k.

Assume that the N-node cluster is connected in a ring topology, where each node has the same bandwidth and computing power. Then, the greedy algorithm is designed in Algorithm 2 to find the approximate optimal scheduling order for the first phase. Algorithm 2 (line 1-2) first initializes P_1 and P_2 based on sequential scheduling. Line 3 calculates the iteration time t_{iter} based on P_1 and P_2. Then, line 4-22 updates P_1 based on the greedy algorithm, which is mainly divided into three stages. The first stage (line 6-10) finds the all-reduce operation that first transfers the basic tensor blocks of layer l. The second stage (line 11-19) raises the

Algorithm 2 Find P_1 for the first phase

Input: $a, b, S_p, L, M[1...L], m[1...L], \tau_b[1...L], t_b[1...L], t_f[1...L]$
Output: $P_1[1...L]$
1: **for** $l = 1 \rightarrow L$ **do**
2: $P_1[l] = l$; $P_2[l] = l$;
3: t_{iter} = CALCULATE_$t_{iter}(P_1, P_2)$;
4: $l = L$;
5: **while** $l \geq 1$ **do**
6: Get $\delta_1, O_{num}, O_{lay}, \tau_{ar}$ from Algorithm 1;
7: **for** $i = 1 \rightarrow \delta_1$ **do**
8: $\tau = 0$;
9: **if** $O_{lay}[i] == l$ **then**
10: $\tau = \tau_{ar}[i] + T_{ar}[m[l]] - \Delta t$;
11: break;
12: **if** $\tau == 0$ **then**
13: $l = l - 1$;
14: continue;
15: **else**
16: **for** $k = l \rightarrow 1$ **do**
17: **if** $k == 1$ **then**
18: P_1^* = ADJUSTPRIORITY(P_1,l,1);
19: **else if** $\tau_b[k] + t_b[k] < \tau \leq \tau_b[k-1] + t_b[k-1]$ **then**
20: P_1^* = ADJUSTPRIORITY(P_1,l,k);
21: break;
22: t_{iter}^* = CALCULATE_$t_{iter}(P_1^*, P_2)$;
23: **if** $t_{iter}^* < t_{iter}$ **then**
24: $t_{iter} = t_{iter}^*$; $P_1 = P_1^*$; $l = k$;
25: $l = l - 1$;
26: **return** P_1;
27: **procedure** ADJUSTPRIORITY(P_1, l, k)
28: $P_1[l] = P_1[k]$
29: **for** $j = k \rightarrow l - 1$ **do**
30: $P_1[j] = P_1[j] + 1$;
31: **return** P_1;
32: **procedure** CALCULATE_$t_{iter}(P_1, P_2)$
33: Get $\tau_f[1...L]$ from Eq. 11;
34: $t_{iter} = \tau_f[L] + t_f[L]$;
35: **return** t_{iter};

priority of the tensor of layer l according to Eq. 13. Note that line 11-12 skips layers that should not be considered since all of their tensors are transmitted in the second phase. The third stage (line 20-22) judges whether raising the priority of the tensor of layer l can shorten t_{iter} and updates P_1.

Algorithm 2 has a time complexity of $O\left(L^3\right)$. It uses an $O\left(L\right)$ search to traverse each layer to determine whether the priority should be adjusted while each priority adjustment requires a time complexity of $O\left(L^2\right)$, so the time complexity is $O\left(L^3\right)$. In particular, Algorithm 2 only needs to be executed once before training, so the overhead of finding the near-optimal scheduling order for the first phase does not affect the training performance.

5 System Implementation

We implement the US-Byte architecture on the framework of ByteScheduler and PyTorch. Fig. 8 shows the overview of the US-Byte architecture. From top to bottom, it consists of (1) user code that describes the DNN and submits DNN training tasks, (2) the API interface of PyTorch frontend, (3) ByteScheduler scheduler that executes communication scheduling, (4) PyTorch engine that determines how to perform the DNN DAG, (5) message-level communication library for all-reduce framework (e.g., NVIDIA NCCL and OpenMPI), and (6) network interface. At the user code level, we design the profiling module. It calculates the priority of the tensor of each layer according to Theorem 1 and Algorithm 2 and converts it into the two-tuples of all all-reduce

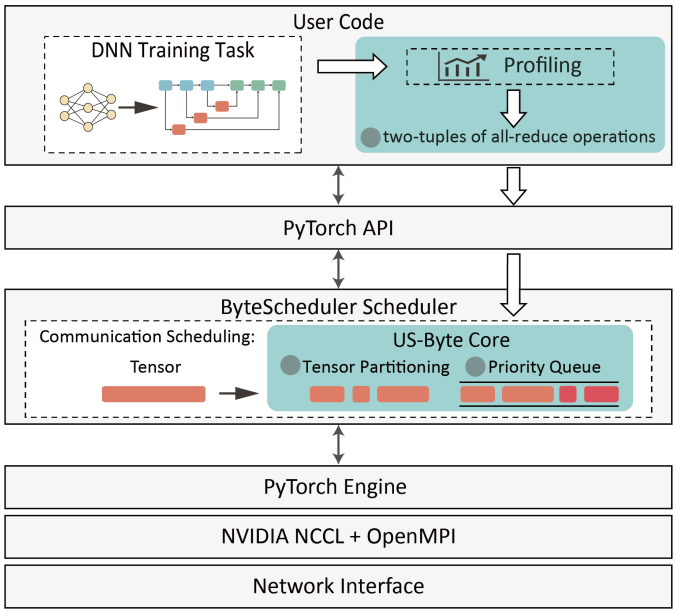


Fig. 8: Overview of the US-Byte architecture. The blue part is added by US-Byte.

operations in one iteration according to the mathematical model of US-Byte. At the ByteScheduler scheduler level, we embed the scheduling framework of US-Byte. In this section, we first elaborate two key points when implementing the US-Byte architecture: (1) the measurement of $t_b^{(l)}$, $t_f^{(l)}$ in the US-Byte model, and (2) the integration of the US-Byte scheduling framework into the ByteScheduler framework. Then, we theoretically compare the energy consumption of US-Byte, ByteScheduler and WFBP during distributed training.

5.1 Time Measurement of Backward Propagation and Forward Propagation

Since the forward propagation process of DNNs is intuitively visible in the class of DNNs, the layer-wise forward propagation time can be measured by Python's time measurement tool. In contrast, PyTorch does not save the gradients of intermediate variables when performing backward propagation (*backward*). Thus, backward propagation of each layer is not visible to the user. In addition, due to the nature of cuda streams, PyTorch can perform different operations on the GPUs simultaneously. Therefore, the gradients of multiple variables in a tensor can be calculated simultaneously when PyTorch performs backward propagation, which makes it theoretically difficult to estimate the layer-wise backward propagation time. To accurately collect the backward propagation time, we use *register_hook* to access the computation of each tensor and use *torch.cuda.synchronize* to synchronize the tensor after it completes the gradient computation. Then, we can collect the layer-wise backward propagation time by calculating the interval between two tensor synchronizations. In addition, to reduce the measurement error, the layer-wise forward propagation and backward propagation time are obtained by averaging the results of 50 measurements.

Algorithm 3 The scheduling process in US-Byte Core

Input: S_p, δ_1, δ_2, $O_{num}[1...\delta_2]$, $O_{lay}[1...\delta_2]$, queue (priority queue of tensor blocks), $Tensor$ (Tensor of each layer from backward propagation)

```
1:  /* Partition a tensor and enqueue tensor blocks. */
2:  procedure PARTITION(Tensor,δ1,δ2,O_num,O_lay)
3:      for i = 1 → δ1 + δ2 do
4:          if O_lay[i] == Tensor.layer then
5:              TensorBlock = Tensor.partition(S_p × O_num[i]);
6:              order = i;
7:              queue.put(order, TensorBlock);
8:  /* Waiting for tensor blocks to finish transmission. */
9:  procedure FINISH(TensorBlock)
10:     channel_is_busy = False;
11: /* Schedule tensor blocks in the queue. */
12: procedure SCHEDULE(δ1,δ2):
13:     _order = 0;
14:     while True do
15:         order, TensorBlock = queue.get();
16:         if not channel_is_busy and order = _order + 1 then
17:             TensorBlock.comm();
18:             channel_is_busy = True;
19:             if order == δ1 + δ2 then
20:                 _order = 0;
21:             else
22:                 _order = _order + 1;
23:         else
24:             Wait until channel_is_busy = False;
```

5.2 Communication Scheduling Process in US-Byte Core

We implement the communication scheduling framework of US-Byte on the ByteScheduler framework. We first obtain P_1 and P_2 by Theorem 1 and Algorithm 2. Then, we calculate the two-tuples of all all-reduce operations through the mathematical model of US-Byte and pass them to the US-Byte core via the PyTorch API.

Algorithm 3 demonstrates the communication scheduling process in the US-Byte core. As backward propagation goes, the tensor of each layer becomes ready one by one. The PARTITION procedure (line 2) partitions the tensor (line 5) and enqueues the tensor blocks into a priority queue (line 7), where the partition size of each tensor block can be obtained through the number of basic tensor blocks merged in each all-reduce operation, and the execution order of the all-reduce operations will be the actual scheduling order of the tensor blocks in the iteration (line 6). The SCHEDULE procedure (line 12) polls the queue constantly (line 15) and transfers each tensor block in order (line 16-22). The tensor blocks can be transmitted when the transmission channel is not busy (line 17). Meanwhile, the transmission channel is released when the transmission of the tensor block is completed (line 10).

Algorithm 3 has a computational complexity of $O\left((L+2)\cdot(\delta_1+\delta_2)\right)$ in one iteration. Specifically, the PARTITION procedure requires L searches of complexity $O\left(\delta_1+\delta_2\right)$ to partition the tensors of each layer, while the computational complexity of both the FINISH and SCHEDULE procedures is $O\left(\delta_1+\delta_2\right)$. Overall, in a cluster, each node needs to execute Algorithms 2 and 3. However, Algorithm 2 only needs to be executed once before training. In addition, Algorithm 3 needs to be executed once in each iteration. Therefore, the total computational overhead of US-Byte is acceptable.

TABLE 2: The Hardware and Software Settings of Clusters.

	Cluster 1	Cluster 2
# of Nodes	8	8
GPU (Nvidia)	Tesla V100S PCIe	RTX 3090
Network	10Gbps	100Gbps
CPU (Intel)	Xeon (Skylake, IBRS)	Xeon(R) Gold 6248R
Memory	256GB	192GB
OS	Ubuntu 18.04	Ubuntu 20.04
Software	CUDA-9.0 OpenMPI-4.0.0 NCCL-2.3.7 PyTorch-1.0.0	CUDA-11.2 OpenMPI-4.0.1 NCCL-2.8.3.1 PyTorch-1.7.0

5.3 Energy consumption analysis

In distributed DNN training, energy consumption can be divided into two primary components: computation energy consumption and communication energy consumption. To theoretically compare the energy consumption of US-Byte, ByteScheduler and WFBP, we assume the same number of iterations for training the same DNN model with the same parameter encoding, dataset, and hardware cluster. By doing so, we can focus on comparing the energy consumption of the three frameworks in one iteration. Since the number of floating-point operations in each forward and backward propagation process remains constant, computation energy consumption is the same across all three frameworks. However, despite the gradient data being transmitted (by the all-reduce primitive in NCCL) remaining the same in each iteration, communication energy consumption still varies among the three frameworks due to differences in communication overheads and scheduler overhead. In one iteration, the numbers of communications in the three frameworks are: $\delta_1+\delta_2$ for US-Byte, $\sum_{i=1}^{L} m^{(l)}$ for ByteScheduler, and L for WFBP. In particular, $L \leq \delta_1+\delta_2 \leq \sum_{i=1}^{L} m^{(l)}$. Furthermore, while US-Byte incurs an additional scheduler overhead in the PARTITION procedure of the US-Byte core compared to ByteScheduler, as evaluated in Table 5, the energy consumption of this overhead is negligible compared to the energy saved by the reduced communication overheads. The actual energy consumption of US-Byte during training is shown in Section 6.6.

6 Evaluation

In this section, we evaluate the performance of US-Byte by real experiments on two 8-node GPU clusters and simulations on a 128-node GPU cluster using six modern DNNs.

6.1 Methodology

Testbed setup. Our testbeds contain two GPU clusters. One is an 8-node NVIDIA Tesla V100S cluster in which each node contains an NVIDIA Tesla V100S GPU card, and the 8 nodes are connected with 10Gbps Ethernet (10GbE). The other is an 8-node NVIDIA RTX 3090 cluster with each node equipped with an NVIDIA GeForce RTX 3090 GPU card, and the 8 nodes are connected with 100Gbps bandwidth. The hardware and software settings of clusters are listed in Table 2.

Benchmarks. We use three well-known datasets. (1) Cifar100 dataset [25], which contains 50,000 training images

TABLE 3: DNNs for Evaluation

Model	# Tensors	# Parameters	Dataset	Batch Size
GoogleNet	59	∼13.0M	Cifar100	128
ResNet-50	161	∼23.7M	ILSVRC12	128
VGG16	32	∼134.7M	ILSVRC12	128
MobileNet-v2	158	∼2.4M	ILSVRC12	32
Inception-v3	292	∼22.0M	ILSVRC12	64
BERT-Base	283	∼109.5M	IMDb	64

TABLE 4: Error between the time of one iteration calculated by US-Byte's model and the actual time of one iteration during training

Model	Cluster 1	Cluster 2
GoogleNet	2.28%	2.69%
ResNet-50	3.55%	2.54%
VGG16	1.22%	1.75%
MobileNet-v2	1.63%	2.15%
Inception-v3	3.41%	4.07%
BERT-Base	4.75%	3.14%

and spans 100 classes; (2) ImageNet dataset (ILSVRC12) [26], which contains 1.28 million training images on 1000 categories; (3) IMDb dataset [27], which contains a set of 25,000 highly polar movie reviews for training.

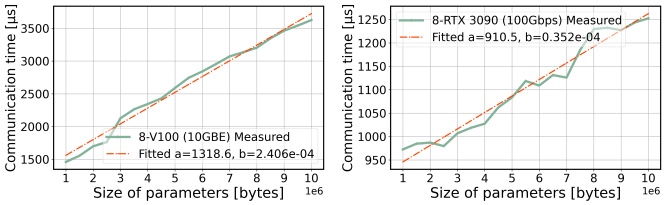
We select six modern DNNs, including five image classification CNN models (GoogleNet [28], ResNet-50 [29], VGG16 [30], MobileNet-v2 [31], Inception-v3 [32]) and one language processing Transformer model (BERT-Base [33]). The training settings of DNNs are listed in Table 3.

Baselines. We compare US-Byte with two representative baselines using the All-reduce architecture. (1) WFBP [17], which is a default optimization implemented on most deep learning frameworks and overlaps gradient aggregation with backward propagation; (2) ByteScheduler [19], the state-of-the-art communication scheduler, which uses tensor partitioning and priority scheduling to overlap gradient aggregation with forward propagation.

We focus on metrics that measure US-Byte performance, such as training speed (images/sec or tokens/sec) and time for one iteration. We compare the training speed of US-Byte and other communication frameworks under different nodes. In addition, we also explore the impact of bandwidth and basic partition size on US-Byte performance. For completeness, we show the non-overlapped communication time and convergence when training ResNet-50 and VGG16. All the reported numbers are averaged over 1000 iterations.

6.2 Measurement of All-Reduce Communication

To verify the model for calculating the time of an all-reduce operation in Eq. 1, we measure the communication time of an All-reduce operation with different size of parameters. Figs. 9a and 9b show the measured time of all-reduce under cluster 1 and cluster 2, respectively. From Fig. 9, we can obtain a linear relationship between the time of an all-reduce operation and the size of parameters. The vertical intercept and slope of the fitted linear function correspond to the values of the parameters a and b, respectively. In addition,



(a) Cluster 1 (b) Cluster 2

Fig. 9: The communication time of all-reduce along with the size of parameters on two different clusters. (a) Cluster 1 with $a = 910.5\mu s$, $b = 0.352 \times 10^{-4}\mu s/byte$; (b) Cluster 2 with $a = 1318.6\mu s$, $b = 2.406 \times 10^{-4}\mu s/byte$.

TABLE 5: Percentage of time overhead for tensor partitioning in one iteration with different communication scheduling frameworks

Model	US-Byte	ByteScheduler	WFBP
GoogleNet	0.93%	0.51%	0%
ResNet-50	1.45%	0.55%	0%
VGG16	0.24%	0.69%	0%
MobileNet-v2	1.23%	0.47%	0%
Inception-v3	1.78%	0.59%	0%
BERT-Base	1.72%	0.66%	0%

parameters a and b can be calculated according to Eq. 14 under different numbers of nodes.

$$a = 2\left(N-1\right)\alpha,\ b = \frac{2\left(N-1\right)}{N}\beta + \frac{\left(N-1\right)}{N}\gamma, \tag{14}$$

where α represents communication overhead between two nodes, β represents transmission time per byte between two nodes and γ represents summation time of two floating point numbers in one node.

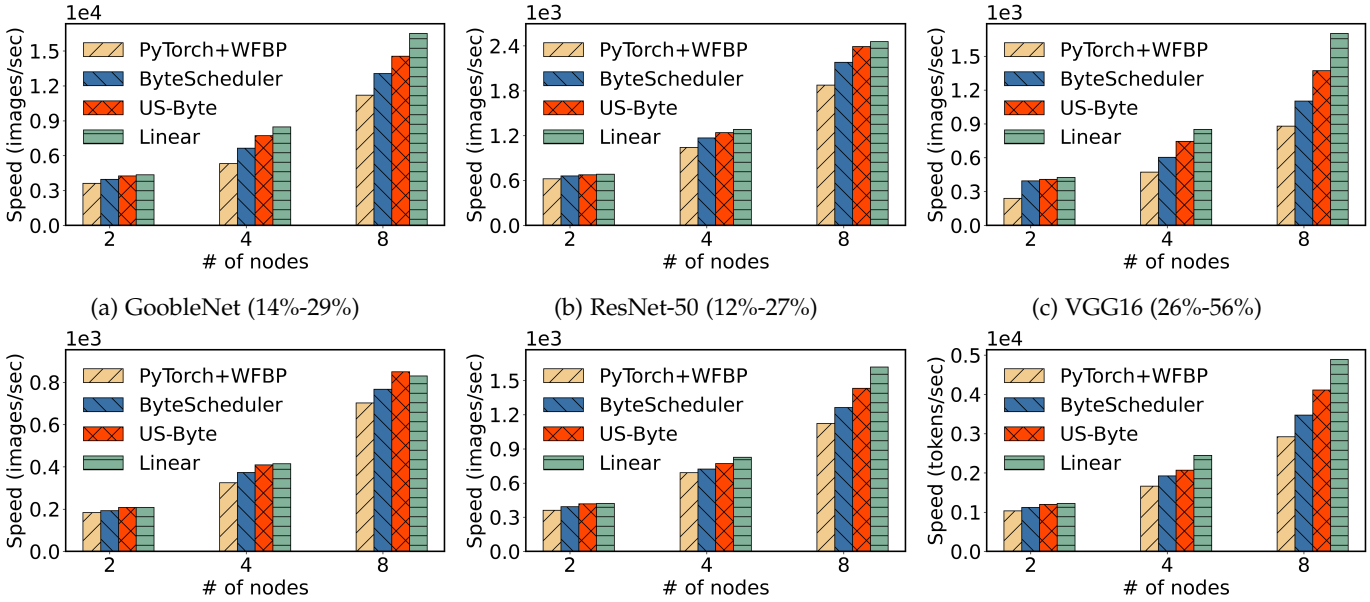
To evaluate the degree to which US-Byte's model accurately mirrors real-world system timings, we measured the error between the time of one iteration calculated by US-Byte's model (theoretical value) and the actual time of one iteration during training (actual value) when training different DNN models with 8 nodes in two clusters, where

$$\text{error} = \frac{|\text{theoretical value} - \text{actual value}|}{\text{actual value}}, \tag{15}$$

and the actual value is averaged over 1000 iterations. The results are presented in Table 4. From Table 4, all errors are less than 5% across six DNN models. Therefore, the degree to which the model mirrors real-world system timings is relatively accurate.

6.3 Worker Number vs. Speed

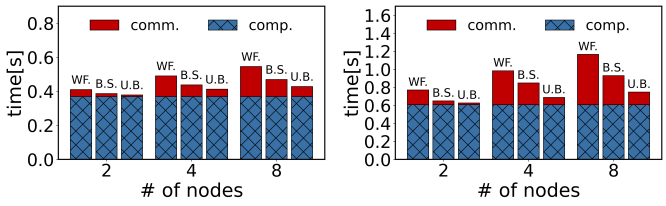
We train six DNNs on Cluster 1 with different numbers of nodes. We obtain the basic partition sizes S_p in US-Byte and ByteScheduler by Bayesian optimization from ByteScheduler. Moreover, we apply credit-based preemption in the ByteScheduler's baseline, and the credit size is also obtained by Bayesian optimization. The training results are shown in Fig. 10. From Fig. 10, US-Byte achieves the best performance on six DNNs and has nearly linear scaling efficiency. US-Byte outperforms ByteScheduler by 11%-26% and WFBP by 21%-56% across the six DNNs in speedup.



(a) GoobleNet (14%-29%) (b) ResNet-50 (12%-27%) (c) VGG16 (26%-56%)

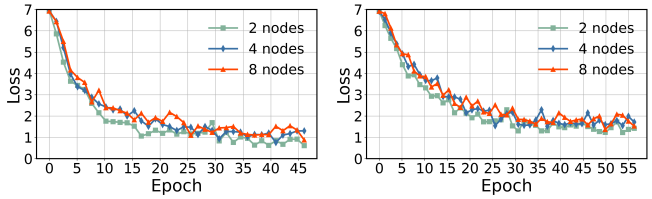
(d) MobileNet-v2 (11%-21%) (e) Inception-v3 (15%-26%) (f) BERT-Base (20%-41%)

Fig. 10: Training speed with different numbers of nodes in Cluster 1. The values in the brackets are the range of improvements of US-Byte over ByteScheduler and WFBP.



(a) ResNet-50 (b) VGG16

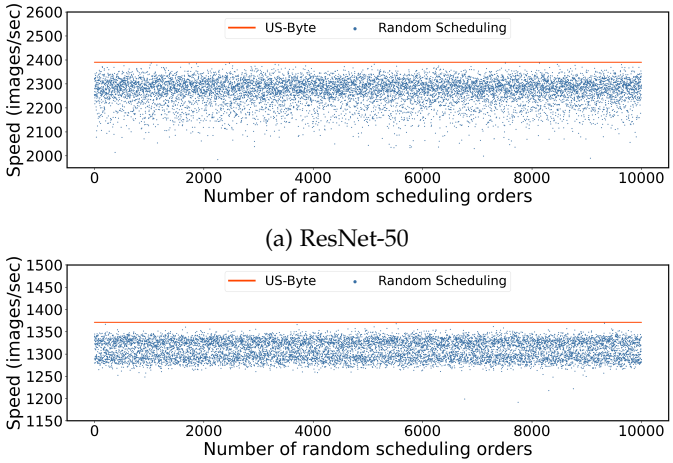
Fig. 11: The time costs of non-overlapped communication and computation in each iteration on Cluster 1. 'WF.', 'B.S.' and 'U.B.' represent WFBP, ByteScheduler and US-Byte frameworks, respectively. 'Comp.' represents the computation time in each iteration and 'Comm.' represents the non-overlapped communication time in each iteration.



(a) ResNet-50 (b) VGG16

Fig. 12: Loss vs. epochs for training ResNet-50 and VGG16 on Cluster 1.

We observe that the acceleration effect of US-Byte on VGG16 and BERT-Base is more pronounced than that of other DNNs. This is because VGG16 and BERT-Base are communication-intensive networks (see Table 3), and the gradient communication time is much longer than the computation time, leaving more optimization space for communication scheduling and benefiting significantly from US-Byte. In contrast, computation-intensive networks such as



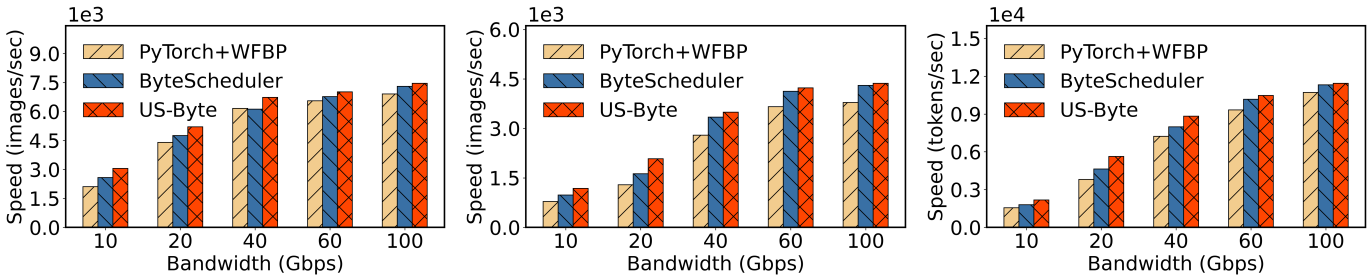
(a) ResNet-50

(b) VGG16

Fig. 13: Training speed with the optimal scheduling order or random scheduling orders in the second phase.

ResNet-50 will obtain better scalability from US-Byte, since they have fewer parameters and the gradient communication time can be almost completely hidden.

To visualize the overlap of gradient communication and gradient computation over different communication frameworks, we measure the time costs of non-overlapped communication and computation in each iteration on Cluster 1 when training ResNet-50 and VGG16 (the time cost of computation = time of forward propagation + time of backward propagation and the time cost of non-overlapped communication time = total time of an iteration - the time cost of computation). The results are shown in Fig. 11. It can be seen that US-Byte achieves better overlap of communication and computing than ByteScheduler and WFBP. Compared with US-Byte, ByteScheduler is difficult to obtain linear scaling



(a) ResNet-50 (b) VGG16 (c) BERT-Base

Fig. 14: Training speed with different bandwidth levels in Cluster 2.

efficiency because the priority scheduling mechanism is destroyed. In addition, when the number of nodes increases, the time costs of non-overlapped communication in WFBP increase significantly due to the larger communication overhead. For completeness, we count the convergence when training ResNet-50 and VGG16 on Cluster 1. Fig. 12 shows that ResNet-50 and VGG16 achieve almost linear statistical performance. Whether 2, 4, or 8 nodes, ResNet-50 can achieve a loss of 1.0 in less than 45 epochs, while VGG16 can achieve a loss of 1.5 in less than 55 epochs.

Scheduler overhead. The US-Byte scheduler overhead is mainly concentrated in the tensor partitioning process. We measure the percentage of time overhead for tensor partitioning in one iteration with different communication scheduling frameworks. The results are shown in Table 5. The time overhead of WFBP is 0 since no tensor partitioning is used. Compared to ByteScheduler, US-Byte has higher time overhead in tensor partitioning. This is because US-Byte needs to search the two-tuples of all-reduce operations to partition each layer's tensor into unequal-sized tensor blocks while ByteScheduler only partitions all tensors into basic tensor blocks based on the basic partition size. Nonetheless, the percentage of scheduler overhead in US-Byte remains negligibly small in one iteration, which is acceptable considering the significant improvement in training speed achieved by US-Byte.

Optimality of the second phase. To verify efficient performance of the scheduling order in the second phased of US-Byte, we compare the training speed when using our proposed optimal scheduling order (based on Theorem 1) and a random scheduling order in the second phase. Fig. 13 shows the measurement results when training ResNet-50 and VGG16, where the blue dots indicate the training speed with 10000 random scheduling orders in the second phase and the red line represents the training speed with our proposed optimal scheduling order in the second phase. From Fig. 13, the training speed with the optimal scheduling order in the second phase is always higher than that with a random scheduling order in the second phase. These experiments demonstrate the efficient performance of our proposed scheduling in the second phase of US-Byte to achieve faster training, and this is consistent with the theoretical analysis in Theorem 1.

6.4 Bandwidth vs. Speed

We investigate the performance gains of US-Byte at different bandwidth levels. We train different DNNs using 8 nodes on Cluster 2 and vary the communication bandwidth between the nodes via the Linux tc tool. The results are presented in Fig. 14. We observe that the training speed remains stable when the network bandwidth is between 60-100Gbps. This is because when the bandwidth is higher and the computing power of the nodes remains constant, the communication time of tensor blocks is reduced, and the communication performance is no longer the bottleneck that limits the scaling efficiency of distributed training. Especially for ResNet-50 with low communication traffic, the communication can be almost completely hidden in the computation time only when the bandwidth is above 40Gbps. US-Byte tends to achieve its highest speedup at a smaller bandwidth level than other frameworks, which benefits from its more efficient overlap of communication and computation.

US-Byte outperforms ByteScheduler by up to 24% for VGG16 at 20Gbps network bandwidth and 21% for BERT-Base at 20Gbps. Even for ResNet-50, US-Byte outperforms ByteScheduler by up to 20% at 10Gbps. When the bandwidth is higher, the speedup by US-Byte, compared to the baselines, may become smaller. US-Byte only outperforms ByteScheduler by 4%-7% and WFBP by 9%-15% at 60Gbps. The time cost of communication is less when the bandwidth level is higher, especially for computation-intensive networks, leaving little room for optimization of US-Byte.

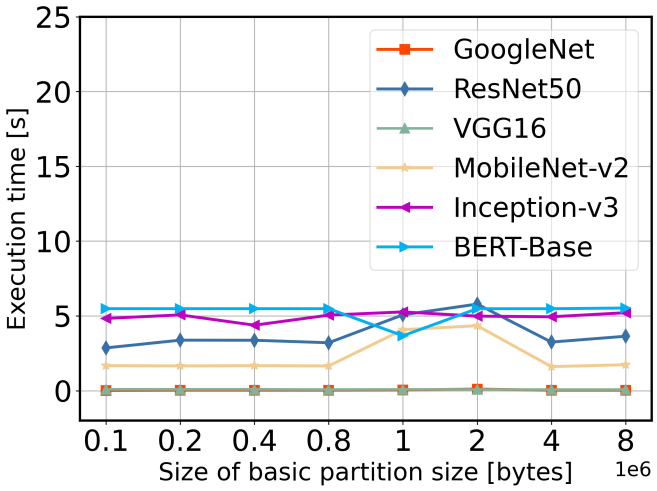
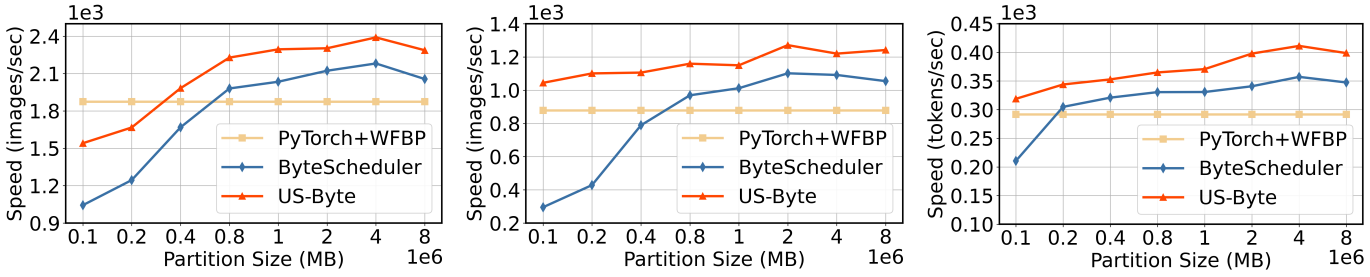
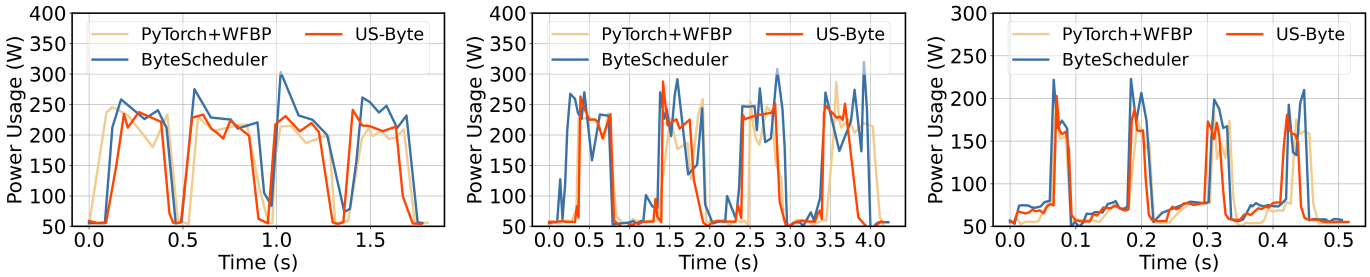


Fig. 15: The execution time of Algorithm 2 with different basic partition sizes.



(a) ResNet-50 (b) VGG16 (c) BERT-Base

Fig. 16: Training speed with different basic partition sizes in Cluster 1.



(a) ResNet-50 (b) VGG16 (c) BERT-Base

Fig. 17: Power usage during training in Cluster 1.

6.5 Partition Size vs. Speed

We further examine the execution time of Algorithm 2 and US-Byte performance gains with different basic partition sizes. We train different DNNs with different S_p from 0.1Mb to 8Mb using 8 nodes on Cluster 2. Figs. 16 and 15 show the results. From Fig. 15, we observe that the execution time of our proposed greedy algorithm remains stable (less than 10s) when S_p changes and only depends on the number of layers of the DNN.

In Fig. 16, the training speed of WFBP remains constant since it is not affected by the change of S_p. However, we observe that the training speed of both US-Byte and ByteScheduler drops significantly when S_p is smaller. This is because when S_p is smaller, the number of tensor blocks is so large that the communication overhead of transmitting tensor blocks cannot be reduced only by communication scheduling. In addition, since US-Byte can adapt to different S_p by merging basic tensor blocks, US-Byte is less affected by the change of S_p than ByteScheduler. On the contrary, although ByteScheduler can increase the credit size to reduce communication overhead, it expands the destruction of the priority scheduling mechanism, which prevents ByteScheduler from obtaining a better performance gain with smaller S_p.

6.6 Energy Consumption

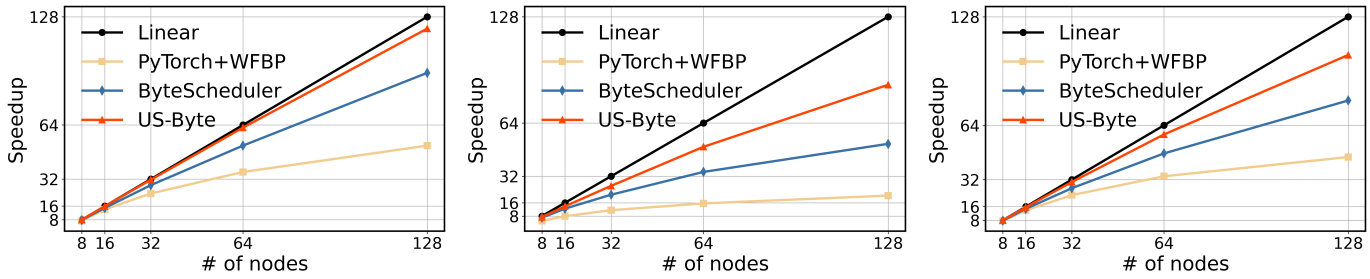
We measure the power usage during distributed training using 8 nodes in Cluster 1 to compare the energy consumption of different communication scheduling frameworks. We monitor the power of node 0 every 5 milliseconds when training three different DNN models and the results are illustrated in Fig. 17.

From Fig. 17, we observe that US-Byte, ByteScheduler and WFBP experience regular peaks and valleys in power usage, and the duration of the cycle is the time of one training iteration. Among the three frameworks, WFBP has the lowest energy consumption, while ByteScheduler consumes the most due to the differences in the number of communication overheads. Furthermore, the energy consumption of US-Byte is much less than that of ByteScheduler and very close to that of WFBP. This is because US-Byte reduces significant communication overheads through tensor fusion as analyzed in Section 5.3.

6.7 Numerical Simulations

Due to the limitation of hardware resources, we conduct simulations to explore the scaling efficiency of US-Byte on a large-scale cluster. The simulation cluster is based on the real single-node GPU and the network performance model. The specific settings of the simulation cluster are as follows: (1) The topology of the simulation cluster is a logical ring, and each node has the same bandwidth. (2) the layer-wise forward propagation and backward propagation time is measured using a real V100 GPU. (3) The values of parameters a and b are obtained through Fig. 9a and Eq. 14 in Cluster 1. (4) We set S_p = 4Mb in US-Byte and ByteScheduler, and set credit size = 2 in ByteScheduler. Then, we simulate US-Byte, ByteScheduler, and WFBP from 8 to 128 nodes.

Overall Performance. Fig. 18 shows the performance comparison of different communication frameworks on the simulated V100 cluster. It is evident that US-Byte has better scaling efficiency than the other two communication frameworks. In the 128-node cluster, US-Byte outperforms ByteScheduler by 28%-69% and WFBP by 234%-332% across the three benchmark models. In particular, US-Byte achieves approximately linear scaling efficiency in ResNet-50. Moreover, the scalability of ByteScheduler is limited by the conflict between credit-based preemption and sequential



(a) ResNet-50 (b) VGG16 (c) BERT-Base

Fig. 18: The performance comparison on the simulated V100 cluster connected with a 10Gbps Ethernet. Baseline of the speedup is on a single V100.

scheduling, which results in the inferior performance of ByteScheduler. WFBP exhibits the worst scaling efficiency among the three communication frameworks. This is because WFBP has a suboptimal bandwidth utilization. When the number of nodes is small (8-32 nodes), the communication overhead is small and can be overlapped in the gradient computation. However, when the number of nodes is large (64-128 nodes), the communication overhead increases so that it cannot be hidden, which makes the scalability of WFBP deteriorate significantly. Instead, US-Byte reduces communication overhead and efficiently overlaps computation and communication by merging the basic tensor blocks and adjusting the scheduling order.

7 Related Work and Discussion

7.1 Related Work

There are various designs to alleviate communication bottlenecks in distributed deep learning. We summarize some related works from two aspects of communication acceleration and communication scheduling.

Communication acceleration: Existing communication acceleration techniques include, but are not limited to: (1) leveraging high throughput and low latency communication links, such as RDMA [34], [35], [36], InfiniBand, Intel Omni-Path, and NVIDIA's NVLink [3]; (2) utilizing message passing interface (MPI) and MPI-like implementations like OpenMPI [4] and Gloo [37]; (3) using high-performance communication collectives, such as NCCL [5] and BLink [38], which support efficient communication between GPUs and many popular deep learning frameworks; (4) reducing data communication during synchronization process, such as gradient quantization, compression and sparsification [39], [40], [41], [42], [43], [44]; (5) using stale parameter updates to reduce the number of synchronization parameters, such as parameter freezing [45], [46], [47], Round-Robin Synchronous Parallel [48] and Bounded Staleness Parallel [49]; (6) tuning deep learning hyper-parameters, such as AutoByte [50]; (7) minimize user-level overhead by conducting parameter aggregation at the transport layer [13]; (8) improving network-layer performance, such as network-level flow scheduling [51], [52] and congestion control [53]. US-Byte is orthogonal to the above approaches and can be combined with them.

Communication scheduling: Due to the layer-wise and tensor-wise structure of DNNs, some works continuously explore to maximize the overlap of communication and computation. In addition to WFBP [17] and ByteScheduler [19], we list some other works. MG-WFBP merges gradient according to the time of backward propagation and communication of each layer to improve communication efficiency [2], [18]. ASC-WFBP utilizes the simultaneous communication of gradients to improve the utilization of network bandwidth [5]. P3 parallelizes gradient communication and computation in MXNet framework with granularity-based priority scheduling [20]. TicTac finds the fixed scheduling order of the tensor with critical-path analysis [21]. PACE transmits tensors based on preemptive communication scheduling and directed acyclic graph of DNN training [4]. Instead, US-Byte schedules unequal-sized tensor blocks in an approximately optimal order, improving bandwidth utilization and minimizing the time of one iteration.

7.2 Discussion

Strengths and limitations. Our proposed US-Byte is an efficient solution to maximize the overlap of communication and computation in distributed deep learning, and the strengths of US-Byte are: (1) dynamically merging a variable number of basic tensor blocks into unequal-sized tensor blocks to reduce communication overheads, (2) scheduling unequal-sized tensor blocks in a near-optimal order to minimize per-iteration time, and (3) adopting the optimal scheduling order of unequal-sized tensor blocks in the second phase.

Despite its strengths, there are still two limitations in US-Byte. First, we model US-Byte to schedule unequal-sized tensor blocks in units of layers, and the basic tensor blocks transmitted at each all-reduce operation are from the same layer. However, we can consider a new model that models in units of basic tensor blocks and merges basic tensor blocks from different layers for transmission. Then, the time of an iteration in the new model will depend on the priority of each basic tensor block. Compared to US-Byte, the new model may further reduce the communication overhead and shorten the time of one iteration. However, the complexity of the new model will be $O\left((\sum_{l=1}^{L} m^{(l)})!\right)$ while the complexity of US-Byte is only $O\left(L!\right)$. In summary, although modeling in units of basic tensor blocks may achieve better

3. https://www.nvidia.com/en-us/design-visualization/nvlink-bridges
4. https://www.open-mpi.org/
5. https://developer.nvidia.com/nccl

performance, its complexity is much higher than US-Byte. As a result, we choose to model US-Byte in units of layers. Second, the scheduling order of unequal-sized tensor blocks and the number of basic tensor blocks in each unequal-sized tensor block (i.e., the two-tuples of all-reduce operations) are determined only once, which depends on a stable cluster environment (including the computing power of nodes and the bandwidth between nodes). Instead, when the cluster environment changes (e.g., new computation or communication tasks are added), automatically updating the two-tuples of all-reduce operations may result in shorter training time.

Greedy Algorithm v.s. Brute Force Search. We have tried to find the optimal scheduling order of unequal-sized tensor blocks for DNNs by brute force search compared to the greedy algorithm. However, the execution time is intolerable. For example, the solution space of VGG16 is $32! \approx 2.6e35$, and according to the actual calculation results, it takes about $1.25s$ to traverse $1e4$ sets of solutions. We thus cannot solve the optimal scheduling order by brute force search. In contrast, our proposed greedy algorithm can obtain a near-optimal solution in less than 1s for VGG16.

Generalizability and portability for different transport protocols. US-Byte has good generalizability and portability for different transportation protocols and significantly outperforms ByteScheduler and WFBP. First, US-Byte is transparent to the transport layer and is generic to different transport protocols. This is because we implement the US-Byte architecture (see Fig. 8) at the user code level and the ByteScheduler scheduler level, which does not depend on specific message-level communication libraries or transport protocols.

Second, US-Byte is more adaptable to different transport protocols. We analyze the impact of changes in latency, throughput and packet loss on different frameworks when using different transport protocols as follows:

(1) Higher latency will result in higher communication overhead. The performance of ByteScheduler will be severely affected due to communication overheads introduced when transmitting basic tensor blocks. Compared to ByteScheduler, US-Byte reduces communication overheads and improves bandwidth utilization by merging basic tensor blocks. Furthermore, the gain gap between US-Byte and WFBP is reduced due to less communication overheads in WFBP. When latency $\rightarrow \infty$, the gains from the three frameworks are US-Byte=WFBP$\gg$ByteScheduler. On the contrary, lower latency will lead to lower communication overhead. The gain gap between US-Byte and ByteScheduler will be narrowed as massive communication overheads are no longer a major bottleneck. However, The performance of WFBP is still limited by new iteration not being started earlier. When latency $\rightarrow 0$, the gains from the three frameworks are US-Byte = ByteScheduler > WFBP.

(2) Lower throughput results in higher transmission time per byte. The performance of WFBP will be severely deteriorated since a lot of time is spent waiting for tensors close to the output layer to be transmitted and new iterations cannot be started. Compared to WFBP, US-Byte schedules unequal-sized tensor blocks in a near-optimal order to begin new iterations earlier, which improves communication efficiency. Meanwhile, since the transmission rate has become the main bottleneck, the gain gap between US-Byte and ByteScheduler has become smaller. When the throughput approaches 0, the gains from the three frameworks are US-Byte=ByteScheduler>WFBP. Instead, higher throughput provides lower transmission time per byte. At this point, the gain gap among the three frameworks is reduced because gradient communication can be almost completely overlapped by gradient computation. When the throughput tends to infinity, the gains from the three frameworks are US-Byte=ByteScheduler=WFBP.

(3) Since we used TCP in our evaluation, it is clear that when packet loss $= 0$, the gains from the three frameworks are US-Byte>ByteScheduler>WFBP. In addition, if packet loss> 0, the transmitted gradient information may be lost, which will affect the model convergence regardless of which communication framework is used. However, SGD-based training has a certain tolerance for packet loss, and packet loss belows a certain fraction (typically 10%-35%) will do little harm to the final model convergence and training time [54], [55]. Therefore, transportation protocols with acceptable packet losses will have a negligible effect on US-Byte or other communication frameworks. When the packet loss becomes higher and seriously affects the model convergence, the gains from the three frameworks are US-Byte=ByteScheduler=WFBP.

8 Conclusion

In this paper, we proposed an efficient communication framework called US-Byte, for scheduling tensor blocks with different partition sizes in a near-optimal order. We built the mathematical model of US-Byte by two phases. We obtained the optimal solution of the second phase by theoretical analysis and designed a low-complexity greedy algorithm to find the approximate optimal solution of the first phase. We implemented US-Byte on PyTorch framework and make it publicly available. US-Byte achieves better scaling efficiency and faster training speed than ByteScheduler and WFBP on extensive experiments with six modern DNNs across two real GPU clusters and a simulation GPU cluster.

Appendix

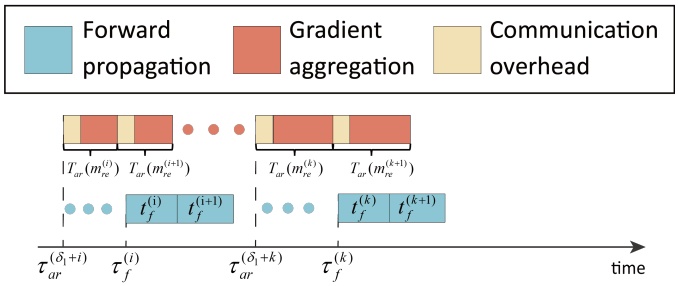
Proof of Theorem 1

When $P_2^{(l)} = l$, without considering Δt, $\tau_f^{(l)}$ can be calculated by

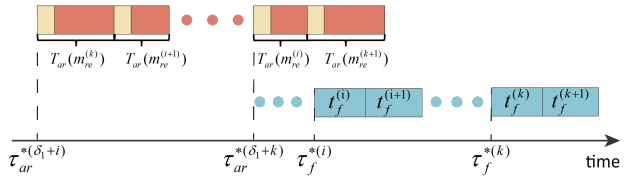
$$\tau_f^{(l)} = \begin{cases} \tau_{ar}^{(\delta_1+l)} + T_{ar}(m_{re}^{(l)}) & l=1 \\ max\{\tau_f^{(l-1)} + t_f^{(l-1)}, \tau_{ar}^{(\delta_1+l)} + T_{ar}(m_{re}^{(l)})\} & l>1 \end{cases}. \tag{16}$$

We use proof by contradiction to prove Theorem 1. We swap the priority of the tensor of layer i and that of layer k based on $P_2^{(l)} = l$, i.e., $P_2^{(i)} = k$ and $P_2^{(k)} = i$, where layer i and layer k are two random layers and satisfy $1 \leq i < k \leq L$. We use the sign $*$ to indicate the new start timestamp for each all-reduce operation and the forward computation of each layer, and no sign $*$ for the original timestamp based on $P_2^{(l)} = l$.

From the example in Fig. 19, we find that after swapping the priority of the tensor of layer i and that of layer k, (1) the



(a) Using sequential scheduling ($P_2^{(l)} = l$).



(b) Swapping the priority of the tensor of layer i and that of layer k based on $P_2^{(l)} = l$.

Fig. 19: An example showing the performance gain of the second phase when using different scheduling orders.

scheduling of the tensor from layer 1 to layer $(i-1)$ does not change, i.e.,

$$\tau_f^{*(l)} = \tau_f^{(l)} \, (1 \le l \le i) \, ; \tag{17}$$

(2) all basic tensor blocks of layer i is transmitted in the $(\delta_1 + k)$-th all-reduce operation, and

$$\tau_{ar}^{*(\delta_1+k)} \ge \tau_{ar}^{*(\delta_1+i)} = \tau_{ar}^{(\delta_1+i)}; \tag{18}$$

(3) the start timestamp of the (δ_1+k+1)-th to the (δ_1+L)-th all-reduce operation remains unchanged, i.e.,

$$\tau_{ar}^{*(\delta_1+i)} = \tau_{ar}^{(\delta_1+i)} \, (k+1 \le i \le L) \, . \tag{19}$$

Then, according to Eq. 16, 17 and 18, we can derive

$$\begin{aligned} \tau_f^{*(i)} &= max\{\tau_f^{*(i-1)} + t_f^{(i-1)}, \tau_{ar}^{*(\delta_1+k)} + T_{ar}(m_{re}^{(i)})\} \\ &\ge max\{\tau_f^{(i-1)} + t_f^{(i-1)}, \tau_{ar}^{(\delta_1+i)} + T_{ar}(m_{re}^{(i)})\} = \tau_f^{(i)}, \end{aligned} \tag{20}$$

i.e., the start timestamp of the forward propagation of layer i is lagged. Meanwhile, all basic tensor blocks of layer $(i+1)$ to layer k have been transferred before $\tau_f^{*(i)}$. Thus, according to Eq. 17, for layer $(i-1)$ to layer $(k-1)$, we have

$$\begin{aligned} \tau_f^{*(l)} &= max\{\tau_f^{*(l-1)} + t_f^{(l-1)}, \tau_{ar}^{*(\delta_1+l)} + T_{ar}(m_{re}^{(l)})\} \\ &= \tau_f^{*(l-1)} + t_f^{(l-1)} \\ &\ge \tau_f^{(l-1)} + t_f^{(l-1)} = \tau_f^{(l)} \end{aligned} \, , \tag{21}$$

and for layer k, we have

$$\begin{aligned} \tau_f^{*(k)} &= max\{\tau_f^{*(k-1)} + t_f^{(k-1)}, \tau_{ar}^{*(\delta_1+i)} + T_{ar}(m_{re}^{(k)})\} \\ &= \tau_f^{*(k-1)} + t_f^{(k-1)} \\ &\ge \tau_f^{(k-1)} + t_f^{(k-1)} = \tau_f^{(k)} \end{aligned} \, . \tag{22}$$

Then, according to Eq. 18 and 21, for layer $(k+1)$ to layer L, we can obtain

$$\begin{aligned} \tau_f^{*(l)} &= max\{\tau_f^{*(l-1)} + t_f^{(l-1)}, \tau_{ar}^{*(\delta_1+l)} + T_{ar}(m_{re}^{(l)})\} \\ &\ge max\{\tau_f^{(l-1)} + t_f^{(l-1)}, \tau_{ar}^{(\delta_1+l)} + T_{ar}(m_{re}^{(l)})\} = \tau_f^{(l)}. \end{aligned} \tag{23}$$

Eventually, according to Eq. 12 and 23, the time of one iteration will become longer due to $\tau_f^{*(L)} \ge \tau_f^{(L)}$. As a consequence, swapping the priorities of the tensor of two random layers based on $P_2^{(l)} = l$ will result in a longer t_{iter}, and sequential scheduling is the optimal scheduling order for the second phase. Theorem 1 is proved.

REFERENCES

[1] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. a. Mao, A. Senior, P. Tucker, K. Yang, Q. Le, and A. Ng, "Large Scale Distributed Deep Networks," in *Adv. neural inf. proces. syst.*, 2012, pp. 1232–1240.

[2] 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.

[3] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch SGD: Training imagenet in 1 hour," *arXiv:1706.02677*, 2017.

[4] Y. Bao, Y. Peng, Y. Chen, and C. Wu, "Preemptive all-reduce scheduling for expediting distributed DNN training," in *Proc. IEEE Conf. Comput. Commun.*, 2020, pp. 626–635.

[5] S. Shi, X. Chu, and B. Li, "Exploiting simultaneous communications to accelerate data parallel distributed deep learning," in *Proc. IEEE Conf. Comput. Commun.*, 2021.

[6] S. Shi, Q. Wang, and X. Chu, "Performance modeling and evaluation of distributed deep learning frameworks on GPUs," in *Proc. IEEE 16th Int. Conf. Dependable Autonomic Secure Comput., 16th Int. Conf. Pervasive Intell. Comput., 4th Int. Conf. Big Data Intell. Comput. Cyber Sci. Technol.*, 2018, pp. 949–957.

[7] L. L. Pilla, "Scheduling Algorithms for Federated Learning With Minimal Energy Consumption," *IEEE Trans. Parallel Distributed Syst.*, vol. 34, no. 4, pp. 1215–1226, 2023.

[8] Z. Quan, Z. Wang, T. Ye, and S. Guo, "Task Scheduling for Energy Consumption Constrained Parallel Applications on Heterogeneous Computing Systems," *IEEE Trans. Parallel Distributed Syst.*, vol. 31, no. 5, pp. 1165–1182, 2020.

[9] Y. Guo, F. Liu, T. Zhou, Z. Cai, and N. Xiao, "Privacy vs. Efficiency: Achieving Both Through Adaptive Hierarchical Federated Learning," *IEEE Trans. Parallel Distributed Syst.*, vol. 34, no. 4, pp. 1331–1342, 2023.

[10] Y. Zhou, Q. Ye, and J. Lv, "Communication-Efficient Federated Learning With Compensated Overlap-FedAvg," *IEEE Trans. Parallel Distributed Syst.*, vol. 33, no. 1, pp. 192–205, 2022.

[11] A. Li, L. Zhang, J. Wang, F. Han, and X. Li, "Privacy-preserving efficient federated-learning model debugging," *IEEE Trans. Parallel Distributed Syst.*, vol. 33, no. 10, pp. 2291–2303, 2022.

[12] R. Saha, S. Misra, A. Chakraborty, C. Chatterjee, and P. K. Deb, "Data-Centric Client Selection for Federated Learning Over Distributed Edge Networks," *IEEE Trans. Parallel Distributed Syst.*, vol. 34, no. 2, pp. 675–686, 2023.

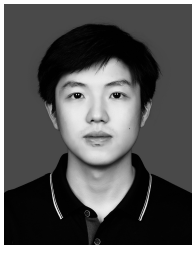
[13] Q. Duan, Z. Wang, Y. Xu, S. Liu, and J. Wu, "Mercury: A Simple Transport Layer Scheduler to Accelerate Distributed DNN Training," in *Proc. IEEE Conf. Comput. Commun.*, 2022, pp. 350–359.

[14] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Adv. neural inf. proces. syst.*, vol. 32, 2019.

[15] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "TensorFlow: a system for Large-Scale machine learning," in *Proc. USENIX Symp. Oper. Syst. Des. Implement.*, 2016, pp. 265–283.

[16] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv:1512.01274*, 2015.

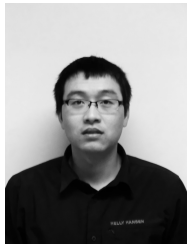
[17] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing, "Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2017, pp. 181–193.
[18] S. Shi, X. Chu, and B. Li, "MG-WFBP: Merging gradients wisely for efficient communication in distributed deep learning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 8, pp. 1903–1917, 2021.
[19] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo, "A generic communication scheduler for distributed DNN training acceleration," in *Proc. ACM Symp. Oper. Syst. Princ.*, 2019, pp. 16–29.
[20] A. Jayarajan, J. Wei, G. Gibson, A. Fedorova, and G. Pekhimenko, "Priority-based Parameter Propagation for Distributed DNN Training," in *Proc. Conf. Machin. Learn. Syst.*, 2019, pp. 132–145.
[21] S. H. Hashemi, S. Abdu Jyothi, and R. Campbell, "TicTac: Accelerating Distributed Deep Learning with Communication Scheduling," in *Proc. Conf. Machin. Learn. Syst.*, 2019.
[22] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in TensorFlow," *arXiv:1802.05799*, 2018.
[23] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B. Su, "Scaling Distributed Machine Learning with the Parameter Server," in *Proc. USENIX Symp. Oper. Syst. Des. Implement.*, 2014, pp. 583–598.
[24] P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations," *J. Parallel Distrib. Comput.*, vol. 69, no. 2, pp. 117–124, 2009.
[25] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.
[26] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2009, pp. 248–255.
[27] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning Word Vectors for Sentiment Analysis," in *Proc. Conf. N. Am. Chapter Assoc. Comput. Linguistics: Hum. Lang. Technol.*, 2011, pp. 142–150.
[28] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2015, pp. 1–9.
[29] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.
[30] A. Z. K. Simonyan, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in *Int. Conf. Learn. Represent.*, 2015.
[31] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 4510–4520.
[32] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 2818–2826.
[33] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. N. Am. Chapter Assoc. Comput. Linguistics: Hum. Lang. Technol.*, 2019.
[34] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, "RDMA over commodity ethernet at scale," in *Proc. ACM Conf. Special Interest Group Data Commun.*, 2016, pp. 202–215.
[35] B. Yi, J. Xia, L. Chen, and K. Chen, "Towards zero copy dataflows using RDMA," in *Proc. SIGCOMM Posters Demos*, 2017, pp. 28–30.
[36] K. Taranov, R. Bruno, G. Alonso, and T. Hoefler, "Naos: Serialization-free RDMA networking in Java," in *USENIX Annu. Tech. Conf.*, 2021, pp. 1–14.
[37] "Gloo," https://github.com/facebookincubator/gloo.
[38] G. Wang, S. Venkataraman, A. Phanishayee, N. Devanur, J. Thelin, and I. Stoica, "Blink: Fast and Generic Collectives for Distributed ML," in *Proc. Conf. Machin. Learn. Syst.*, 2020, pp. 172–186.
[39] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding," in *Adv. neural inf. proces. syst.*, 2017.
[40] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li, "Terngrad: Ternary gradients to reduce communication in distributed deep learning," in *Adv. neural inf. proces. syst.*, vol. 30, 2017.
[41] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep gradient compression: Reducing the communication bandwidth for distributed training," *arXiv:1712.01887*, 2017.
[42] S. Shi, Q. Wang, K. Zhao, Z. Tang, Y. Wang, X. Huang, and X. Chu, "A distributed synchronous SGD algorithm with global top-k sparsification for low bandwidth networks," in *Proc. 39th IEEE Int. Conf. Distrib. Comput. Syst.*, 2019, pp. 2238–2247.
[43] S. Shi, K. Zhao, Q. Wang, Z. Tang, and X. Chu, "A Convergence Analysis of Distributed SGD with Communication-Efficient Gradient Sparsification," in *Int. Joint Conf. Artif. Intell.*, 2019, pp. 3411–3417.
[44] A. F. Aji and K. Heafield, "Sparse communication for distributed gradient descent," in *Conf. Empir. Methods Nat. Lang. Process.*, 2020, pp. 172–186.
[45] C. Chen, H. Xu, W. Wang, B. Li, B. Li, L. Chen, and G. Zhang, "Communication-Efficient Federated Learning with Adaptive Parameter Freezing," in *Proc. 41st IEEE Int. Conf. Distrib. Comput. Syst.*, 2021, pp. 1–11.
[46] W. Luping, W. Wei, and L. Bo, "CMFL: Mitigating communication overhead for federated learning," in *Proc. 39th IEEE Int. Conf. Distrib. Comput. Syst.*, 2019, pp. 954–964.
[47] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G. R. Ganger, P. B. Gibbons, and O. Mutlu, "Gaia: Geo-Distributed Machine Learning Approaching LAN Speeds," in *Proc. USENIX Symp. Networked Syst. Des. Implement.*, 2017, pp. 629–647.
[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] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server," in *Adv. neural inf. proces. syst.*, 2013.
[50] Y. Ma, H. Wang, Y. Zhang, and K. Chen, "AutoByte: Automatic Configuration for Optimal Communication Scheduling in DNN Training," in *Proc. IEEE Conf. Comput. Commun.*, 2022, pp. 760–769.
[51] S. Wang, D. Li, J. Zhang, and W. Lin, "Cefs: compute-efficient flow scheduling for iterative synchronous applications," in *Proc. Int. Conf. Emerg. Netw. EXp. Technol.*, 2020, pp. 136–148.
[52] 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.
[53] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh *et al.*, "HPCC: High precision congestion control," in *Proc. Conf. ACM Spec. Interest Group Data Commun.*, 2019, pp. 44–58.
[54] J. Xia, G. Zeng, J. Zhang, W. Wang, W. Bai, J. Jiang, and K. Chen, "Rethinking transport layer design for distributed machine learning," in *Proc. Asia-Pacific Workshop on Networking*, 2019, pp. 22–28.
[55] Z. Chen, L. Shi, X. Liu, X. Ai, S. Liu, and Y. Xu, "Boosting distributed machine learning training through loss-tolerant transmission protocol," *arXiv:2305.04279*, 2023.



Yunqi Gao (Student Member, IEEE) received the BEng degree in Electronic Information Engineering from the Northeastern University of China, in 2021. He is currently pursuing the Ph.D. degree with the College of Information Science and Electronic Engineering, Zhejiang University, Hangzhou, China. His research interests include parallel computing, distributed systems and machine learning.

Bing Hu (Senior Member, IEEE) received the BEng and MPhil degrees in Communications Engineering from the University of Electronic Science and Technology of China, in 2002 and 2005, respectively, and the PhD degree from the Department of Electrical and Electronic Engineering, University of Hong Kong, in 2009. In 2010, he joined the College of Information Science and Electronic Engineering, Zhejiang University, where he is currently an associate professor. His research interests include datacenter networks, AI networks, deterministic networks, and 6G.

Mahdi Boloursaz Mashhadi (Senior Member, IEEE) is a Lecturer at the 5G/6G Innovation Centre (5G/6GIC) at the Institute for Communication Systems (ICS), University of Surrey (UoS), UK. Prior to joining ICS, he was a postdoctoral research associate at the Intelligent Systems and Networks (ISN) Research Group, Imperial College London, 2019-2021. He received B.S., M.S., and Ph.D. degrees in mobile telecommunications from the Sharif University of Technology (SUT), Tehran, Iran, in 2011, 2013, and 2018, respectively. He was a visiting research associate with the University of Central Florida, Orlando, USA, in 2018, and Queen's University, Ontario, Canada, in 2017. He has more than 40 peer-reviewed publications and patents in the areas of wireless communications, machine learning, and signal processing. He received the Best Paper Award from the IEEE EWDTS 2012 conference, and the Exemplary Reviewer Award from the IEEE ComSoc in 2021 and 2022. He has served as a panel judge for the International Telecommunication Union (ITU) to evaluate innovative submissions on applications of AI/ML in 5G and beyond wireless networks since 2021. He is an associate editor for the Springer Nature Wireless Personal Communications Journal.



A-Long Jin (Student Member, IEEE) received his Ph.D. degree from the Department of Electrical and Electronic Engineering, The University of Hong Kong, in 2023. Before that he received the B.Eng. degree in communications engineering from Nanjing University of Posts and Telecommunications, China, in 2012, and the M.Sc. degree in computer science from University of New Brunswick, Fredericton, Canada, in 2015. His research interests include computer networks, distributed systems, and machine learning.

Pei Xiao (Senior Member, IEEE) is currently the technical manager of 5GIC/6GIC, leading the research team in the new physical layer work area, and coordinating/supervising research activities across all the work areas (https://www.surrey.ac.uk/institute-communication-systems/5g-6g-innovation-centre). Prior to this, he worked at Newcastle University and Queen's University Belfast. He also held positions at Nokia Networks in Finland. He has published extensively in the fields of communication theory, RF and antenna design, signal processing for wireless communications, and is an inventor on over 15 recent 5GIC patents addressing bottleneck problems in 5G systems.

Chunming Wu (Senior Member, IEEE) received the Ph.D. degree in computer science from Zhejiang University, Hangzhou, China, in 1995.

He is currently a Professor with the College of Computer Science and Technology, Zhejiang University. His research interests include software defined networks, proactive network defense, network virtualization, and intelligent networks.