DeAR: Accelerating Distributed Deep Learning with Fine-Grained All-Reduce Pipelining

Lin Zhang[†], Shaohuai Shi[‡*], Xiaowen Chu[§†], Wei Wang[†], Bo Li[†], Chengjian Liu[¶]
[†]The Hong Kong University of Science and Technology, [‡]Harbin Institute of Technology, Shenzhen,
[§]The Hong Kong University of Science and Technology (Guangzhou), [¶]Shenzhen Technology University
lzhangbv@connect.ust.hk, shaohuais@hit.edu.cn, xwchu@ust.hk, {weiwa, bli}@cse.ust.hk, liuchengjian@sztu.edu.cn

***Abstract*—Communication scheduling has been shown to be effective in accelerating distributed training, which enables all-reduce communications to be overlapped with backpropagation computations. This has been commonly adopted in popular distributed deep learning frameworks. However, there exist two fundamental problems: (1) excessive startup latency proportional to the number of workers for each all-reduce operation; (2) it only achieves sub-optimal training performance due to the dependency and synchronization requirement of the feed-forward computation in the next iteration. We propose a novel scheduling algorithm, DeAR, that decouples the all-reduce primitive into two continuous operations, which overlaps with both backpropagation and feed-forward computations without extra communications. We further design a practical tensor fusion algorithm to improve the training performance. Experimental results with five popular models show that DeAR achieves up to 83% and 15% training speedup over the state-of-the-art solutions on a 64-GPU cluster with 10Gb/s Ethernet and 100Gb/s InfiniBand interconnects, respectively.**

I. Introduction

Training a complex deep neural network (DNN) model over a large data set requires a massive amount of compute resources and is typically performed on a cluster of GPU machines [1]–[3]. To accelerate distributed training, many different ways of parallelism have been proposed recently, such as data-parallel [4], model-parallel [1], pipeline-parallel [5], and the combination of the above [6]. Among them, the data-parallel synchronous stochastic gradient descent (S-SGD) algorithms are the most popular when each worker machine has sufficient GPU memory to hold the training model. In S-SGD, the training data is sharded across multiple GPU workers. Each worker iteratively updates the training model by aggregating the local gradients computed with local data samples. To efficiently support gradient aggregation, current training frameworks use the all-reduce architecture [2,4,7]–[10], in which gradient aggregation is performed with an all-reduce collective. The all-reduce architecture has been widely adopted in practice to distributed training, according to the MLPerf training benchmarks[1].

As the model size and the number of workers increase, gradient aggregation requires extensive data communications, which easily become the bottleneck [11,12]. System-level optimizations are thus needed to address this scalability issue. One effective approach that exploits the layer-wise structure of DNN models is to pipeline gradient calculation (computing tasks) with gradient aggregation (communication tasks) in the backpropagation stage, so as to hide the communication overhead and thus improve the system throughput [13,14]. This approach, known as wait-free backpropagation (WFBP) [13], has been implemented as the default mechanism in modern deep learning (DL) frameworks such as TensorFlow, PyTorch-DDP [15], and Horovod [16,17]. However, WFBP only pipelines communications with gradient computations in the backpropagation stage, which does not consider the feed-forward stage, thus making it sub-optimal. It is worth pointing out that feed-forward computations account for around one third of the total computation time in each iteration [18], which can be properly exploited to further accelerating the training speed.

However, it is challenging to enable pipelining between the communication tasks for gradient aggregation and the next iteration's feed-forward computing tasks under the all-reduce architecture, for two reasons. First, a tensor's gradient aggregation is an all-reduce primitive, which can only begin after its gradient has been calculated in backpropagation and should be synchronized before the next iteration's feed-forward computation. Thus, it only allows coarse-grained scheduling between communications and computations. Second, the all-reduce communication tasks are coming in a first-in, first-out (FIFO) order with the dependency of backpropagation computing tasks. Communication tasks can be re-ordered to be pipelined with feed-forward computing tasks. Yet, different workers execute the computing tasks concurrently, such a re-ordering needs to be done collectively in a consistent manner by all workers to ensure the correctness of all-reduce results. Therefore, this requires synchronization among workers in each iteration, which causes extra communication overheads.

To address the two challenges above, we propose a new scheduling algorithm called DeAR[2] that decouples the all-reduce primitive to two operations, so as to enable fine-grained scheduling without introducing extra communication overhead. DeAR applies three novel techniques to distributed training for the all-reduce architecture. To the best of our knowledge, we are the first to decouple the all-reduce primitive without introducing extra time costs so that communications

*Corresponding author.

[1]https://mlcommons.org/en/training-normal-11/

[2]Source code can be found in https://github.com/lzhangbv/dear_pytorch.

become possible to be overlapped with feed-forward computations in distributed training.

First, though the all-reduce operation is a primitive in distributed training, all-reduce implementations can be handled as a combination of basic routines [19]–[22]. For example, a classic implementation of the widely used ring-based all-reduce is a combination of a reduce-scatter collective followed by an all-gather collective [19,20]. Based on the nature of all-reduce implementations, we decouple the all-reduce primitive to two continuous collectives in distributed training, which allows a fine-grained schedule of communication tasks.

Second, given that one all-reduce primitive is decoupled into two operations, we propose to schedule the first operation to be pipelined with backpropagation computing tasks, and the second operation pipelined with feed-forward computing tasks. By doing so, there is no need to re-order the communication tasks while enabling the pipelining between the communication tasks and all the computing tasks without introducing any extra communication overhead during training.

Third, due to the pipelining between the communication tasks and feed-forward computing tasks, tensor fusion techniques [16,23,24], which have been proven effective in reducing the latency overhead in WFBP [13], becomes impractical in DeAR. The main challenge is how to determine which tensor should (not) be fused. To this end, we propose a dynamic tensor fusion algorithm using Bayesian optimization in DeAR to judiciously determine which tensors should be fused to improve the training efficiency, without any prior knowledge about the model and cluster configurations.

We implemented DeAR atop PyTorch. Our implementation provides an easy-to-use API such that users can integrate our training algorithm by adding a few lines of code. Extensive experiments are conducted with popular DNNs on a 64-GPU cluster under various system configurations. Experimental results show that, compared with the state-of-the-art solutions, including PyTorch-DDP, Horovod, MG-WFBP [23], and ByteScheduler [25], DeAR accelerates the model training by up to 83% and 15% over 10Gb/s Ethernet and 100Gb/s InfiniBand interconnects, respectively. In all experiments, the training speedup enabled by DeAR reaches 72.3-99.2% of the maximum possible.

II. Background and Motivation

A. Mini-batch SGD

The training of DNN models is to minimize a designed loss function $\mathcal{L}(w, X)$, where w is the model parameter and X is the training data. In mini-batch SGD, the model parameters is updated iteratively based on its first-order gradient. Specifically, at each iteration i, a mini-batch data (X_i) is randomly sampled to calculate the loss through feed-forward from the first layer to the last layer; and then the first-order gradient w.r.t. the model parameter is calculated through backpropagation. Then, the gradient is used to update the parameter. Formally, the update formula at the i^{th} iteration can be represented as follows.

$$w_{i+1} = w_i - \eta \nabla \mathcal{L}(w_i, X_i), \tag{1}$$

where η is the learning rate, w_i and X_i are the model parameter and sampled data at iteration i, respectively. Thus, in a single-GPU environment, the training time is mainly consumed in the feed-forward and backpropagation computing tasks.

B. S-SGD

When exploiting multiple workers (e.g., GPUs) to train a single model, synchronous SGD (S-SGD) with data parallelism is a de-facto approach for training as it preserves the convergence properties of mini-batch SGD. In S-SGD, each iteration's training data X_i is distributed to P workers as X_i^p at worker p, where $p = 1, 2, ..., P$ on a P-worker cluster, and all workers keep consistent parameters at every iteration. The update rule of S-SGD is

$$w_{i+1} = w_i - \eta \frac{1}{P} \sum_{p=1}^{P} \nabla \mathcal{L}(w_i, X_i^p). \tag{2}$$

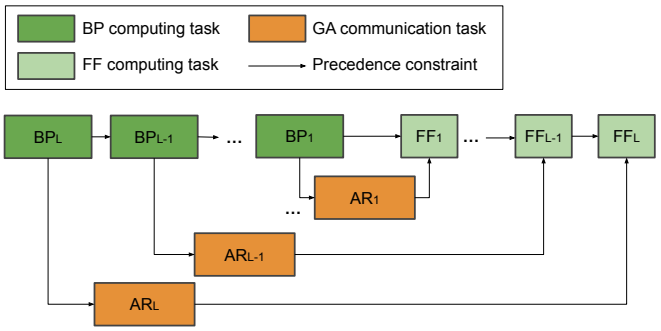
It is seen that the distributed gradients should be aggregated before updating the model parameter, which introduces communication costs and limits the system scaling efficiency. In practice, the gradient aggregation (GA) can be implemented through a parameter server [26] or an all-reduce collective. We focus on the all-reduce implementation in this work. In summary, the iteration time of S-SGD contains the feed-forward computation time, the backpropagation computation time, and the communication time of gradient aggregation.

Due to the layer-wise structure of DNN models, the computing tasks and communication tasks can be organized as a directed acyclic graph (DAG) as shown in Fig. 1(a). One layer's communication (AR_l) can only begin after its gradient has been calculated (BP_l), and its feed-forward computation (FF_l) should wait for the completion of AR_l. According to the DAG, it is possible to schedule the order of different tasks so that they can be overlapped to shorten the iteration time.

C. Wait-free backpropagation

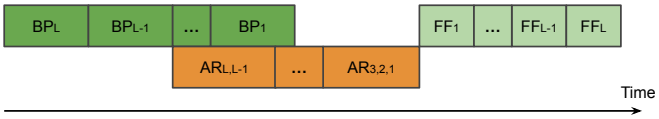
Gradient aggregation of some layers can be overlapped with backpropagation using the wait-free backpropagation algorithm (WFBP) [13,14], in which the gradient communication can immediately begin after the gradient is calculated. Due to the nature of backpropagation, where the gradients are calculated from the last layer to the first layer, multiple layers communications are scheduled with a first-in-first-out (FIFO) order as shown in Fig. 1(b). In modern DNN models, there are many layers which have a relatively small number of gradients that need to be aggregated, thus WFBP requires tensor fusion (e.g., MG-WFBP [23]) to alleviate the startup overhead in all-reduce communications as shown in Fig. 1(c).

However, WFBP and its variant only allow the gradient aggregation communication tasks to be pipelined with backpropagation computing tasks as shown in Fig. 1(b)(c). That

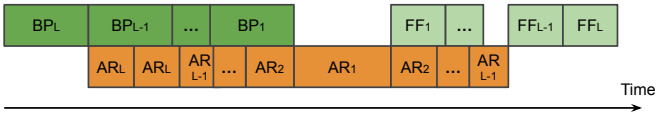


(a) DAG of computing and communication tasks.

(b) WFBP.



(c) WFBP with tensor fusion.



(d) ByteScheduler: priority scheduling and tensor partitioning.

Fig. 1: (a) The DAG of computing and communication tasks in an L-layer DNN, and (b-d) the timeline of S-SGD algorithms with different schedules. (b) WFBP: Gradient communication of each layer begins after that layer's gradients are calculated and the communications are executed in a FIFO order. (c) The gradients of nearby layers are fused to be communicated together. (d) ByteScheduler: large tensors may be partitioned into multiple smaller tensors and the order of communications is based on their priorities but not FIFO.

is, the feed-forward computing tasks of the next iteration can only begin after all the GA communication tasks of the current iteration have been completed. Thus, the communication tasks have no opportunity to be pipelined with the feed-forward computing tasks, which is sub-optimal if the communication time cannot be fully hidden by the backpropagation computation time. The feed-forward computing tasks normally consume around one to third of the total computation time at each iteration [18]. If one can pipeline the communication tasks with feed-forward computing tasks, the one to third computation time could also be saved.

D. Priority scheduling and tensor partitioning

Though ByteScheduler regards the gradient aggregation as a pair of (PUSH, PULL) in the PS architecture to enable a finer-gained schedule, it cannot use the (PUSH, PULL) feature in all-reduce which is a primitive in existing deep learning frameworks. Instead, to enable some communication tasks to be overlapped with feed-forward computing tasks, ByteScheduler [25] re-orders the communication tasks and issues the tasks in an "optimal" order by allowing large tensors to be partitioned into multiple small tensors as shown in Fig. 1(d). First, the communication of the second layer (i.e., AR_2), which can only begin after AR_L to AR_3 in a FIFO order, is scheduled to be executed prior to AR_{L-1}. Second, some large tensors may be partitioned into multiple small tensors to provide finer-grained scheduling. For example, the tensor of layer 2's gradient is partitioned to two tensors which can be separately completed with two all-reduce operations. The priority scheduling and tensor partitioning techniques in enabling the communication tasks to be pipelined with feed-forward computing tasks may work well in the PS architecture [25], but it would have significant performance issues in the all-to-all architecture due to the following two problems.

First, re-ordering the communication tasks requires all workers, which execute computing tasks concurrently during training, to have a consensus on communicating a particular tensor. In other words, before aggregating the gradient of a layer, all workers should negotiate with each other that the layer is ready for communicating to ensure the correctness of training. This would introduce extra communication overheads. Even though the negotiation only needs to communicate several bytes of data, it may have significant latency with the increasing number of workers, especially on high-latency interconnects (e.g., 10Gb/s Ethernet).

Second, using tensor partitioning for a finer-grained schedule may introduce extra startup overheads of communications. Generally, the time cost of an all-reduce communication contains a startup overhead that is proportional to the number of workers [20]–[23]. For example, in the widely used ring-based all-reduce algorithm, which is a default in NCCL, the startup time is linear to the number of GPUs [21]. Therefore, partitioning a tensor to n smaller tensors to be communicated separately would introduce extra $n-1$ startup overheads. For example, on a 64-GPU cluster with 10Gb/s Ethernet, all-reducing a 1MB message takes around 4.5ms, while all-reducing a 500KB message takes around 3.9ms.

In summary, existing scheduling techniques to enable the pipelining between communication tasks and feed-forward computing tasks are impractical for distributed training in the all-to-all architecture. Pipelining the communication tasks with feed-forward computing tasks is expected to save one to third of the computation time, but the introduced extra communication overhead in ByteScheduler may be larger than the hidden computation time, resulting an even worse performance.

This motivates us to decouple the all-reduce primitive based on its implementation nature to two operations for a finer-grained schedule, and it does not introduce any extra communication overhead.

III. DeAR: Decoupling the all-reduce primitive

The design philosophy of our DeAR is to decouple the all-reduce primitive to two continuous operations without introducing extra communication overheads.

A. Decoupling all-reduce with zero overhead

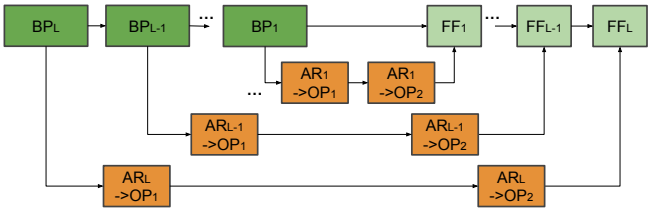
According to the inherent feature of all-reduce primitive that tries to maximally utilize the network bandwidth or minimally reduce the latency [21,27], it should be implemented with multiple rounds of communications, each of which has multiple workers participating in sending and receiving messages simultaneously. Thus, it is very common that the all-reduce algorithm is implemented with a combination of other basic routines [20]. For example, the ring-based all-reduce algorithm can be implemented by a ring-based reduce-scatter operation followed by a ring-based all-gather operation [21]. Thus, theoretically, the all-reduce primitive can be decoupled into two or more continuous operations whose total time equals to the time cost of the all-reduce primitive. The decoupled operations of a primitive will allow finer-grained tasks scheduling in distributed training.

As our goal is to enable some communication tasks to be pipelined with feed-forward computing tasks, we break down the all-reduce operation OP_{ar} into two continuous communication operations, say OP_1 and OP_2. Note that the total time of OP_1 and OP_2 equals to the time of OP_{ar}, which means the decoupling is free. Therefore, OP_1 of different layers can still be pipelined with backpropagation computing tasks, while OP_2 can be pipelined with feed-forward computing tasks. The DAG of computing and communication tasks in DeAR is shown in Fig. 2(a). One layer's gradient aggregation is composed of two continuous communication operations. Compared to Fig. 1(a), the fine-grained DAG with decoupled all-reduce allows us to schedule OP_1 and OP_2 communication tasks separately, which offers great opportunities to pipeline the communication tasks with feed-forward computing tasks without tensor partitioning.

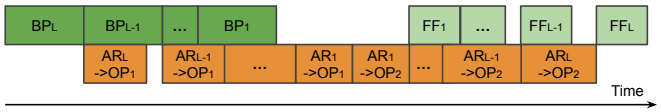
In this work, we use the ring-based all-reduce algorithm, which is widely used in distributed training, as an example to show how we decouple it with zero overhead. Note that the key idea of DeAR can be applied in any all-reduce algorithms as long as they can be decoupled into two operations without introducing any extra overhead. In the ring-based algorithm on a P-worker cluster, the d elements are divided to P chunks, each of which has d/P elements. In the first step, each chunk will be reduced to each worker via $P-1$ communication rounds, which is a reduce-scatter operation and it takes a time complexity of

$$t_{rs} = (P-1)(\alpha + \frac{d}{P}\beta), \tag{3}$$

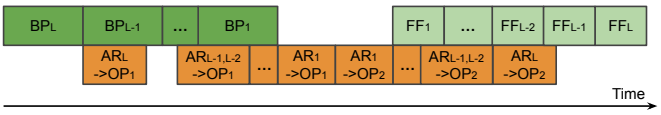
where α and β are the latency and transmission time per element between two workers according to the $\alpha-\beta$ cost model [28]. As we only focus on the communication time, we omit the overhead of arithmetic operations of accumulating elements in Eq. 3.



(a) DAG of computing and communication tasks in DeAR.



(b) Timeline of DeAR with an order of FIFO for communications.



(c) Timeline of DeAR with tensor fusion.

Fig. 2: (a) The DAG of computing and communication tasks with an L-layer DNN in DeAR. (b) DeAR without tensor fusion: the decomposed communications are executed in a FIFO order. (c) DeAR with tensor fusion: Nearby gradients could be merged to a single one for the decoupled operations.

In the second step, each reduced chunk at every worker is broadcast to all other workers, which is an all-gather operation and it also takes $P-1$ communication rounds in the ring-based algorithm. The all-gather operation has a time complexity of

$$t_{ag} = (P-1)(\alpha + \frac{d}{P}\beta). \tag{4}$$

Putting Eq. 3 and Eq. 4 together, we achieve the time complexity of an all-reduce operation as follows.

$$t_{ar} = 2(P-1)\alpha + \frac{2(P-1)d}{P}\beta. \tag{5}$$

B. Pipelining communication tasks without re-ordering

By decoupling the all-reduce primitive to two continuous operations, it becomes possible to pipeline the first communication operation with backpropagation computing tasks, and pipeline the second communication operation with feed-forward computing tasks as shown in 2(b). To guarantee data dependencies between tasks at run-time, we propose 1) BackPipe: starting the communication task of OP_1 immediately when the gradient of one layer is ready in the backward pass, and 2) FeedPipe: waiting for the completion of the communication task of OP_2 of one layer before its feed-forward computation, and starting the communication task of OP_2 of the next layer. Besides, we synchronize all the tasks of OP_1 at the end of BackPipe to ensure the dependencies between OP_1 and OP_2.

In doing so, our DeAR can execute the communication tasks asynchronously to support pipelining with both feed-forward and backpropagation computing tasks, while preserving data

dependencies between tasks without any requirement to adjust the order of communication tasks. That is, communication tasks are issued among all workers consistently from the last layer to the first layer during backpropagation and its reverse order during feed-forward, respectively. Therefore, all workers do not need the time-consuming negotiation with each other to reach a consensus in which tensors should be aggregated.

In summary, compared with the WFBP [13,14] or its variant [16,23] algorithms, DeAR is able to overlap the gradient aggregation communications with both feed-forward and backpropagation computing tasks. Compared with ByteScheduler [25], DeAR enables a finer-grained tasks schedule in distributed training without the requirement of partitioning tensors and re-ordering the communication tasks.

Moreover, DeAR reserves the property of tensor fusion as like WFBP, where the gradients in nearby layers can be merged together to be communicated once to reduce the startup overhead. Unlike ByteScheduler which exploits tensor partitioning (a mutual operation with tensor fusion) to provide a fine-grained schedule, DeAR schedules OP_1 with backpropagation computing tasks and OP_2 with feed-forward computing tasks, which means OP_1 in different layers are possible to be merged to be communicated together and it is similar to OP_2. We discuss the details about tensor fusion in the following section.

IV. Tensor fusion in DeAR

Tensor fusion [16,23] has been proven to be a simple yet effective approach to reducing the startup overheads of all-reduce operations. It has become a default feature in distributed DL frameworks like PyTorch-DDP [15] and Horovod [16]. However, how to determine which layers should be merged is quite challenging as merging any nearby layers requires to wait for their backpropagation computing tasks to be completed.

A. Preliminary of tensor fusion techniques

In the standard all-reduce primitive case (like PyTorch-DDP and Horovod), where the communication tasks only overlap with backpropagation computing tasks, a buffer with a pre-defined size (e.g., 25MB in PyTorch-DDP and 64MB in Horovod) is allocated to store the ready-to-communicate tensors. When the total size of ready tensors reaches out of the buffer size, the buffer is communicated for aggregation with an all-reduce operation, which means multiple tensors stored in the buffer only communicate once at each iteration. The buffer data is then copied back to the original tensors when the current all-reduce operation completes. Due to the gradients in different layers become ready in a backward order, fusing any two layers needs to wait for the completion of all gradients in these two layers as shown in Fig. 1(c). Thus, it is non-trivial to determine the optimal buffer size to achieve minimal iteration time.

One can also measure the backpropagation computation time of each layer and estimate the communication time of all-reduce to dynamically determine whether the benefit of merging any two nearby layers is larger than the sacrifice of the waiting time [23]. Yet, two main issues may make the solution impractical [23]. First, the layer-wise backpropagation time is quite difficult to be correctly measured as each layer gradient may be computed asynchronously. Second, the variant tensor sizes in a DNN model make it difficult to predict the communication time accurately by a simple model.

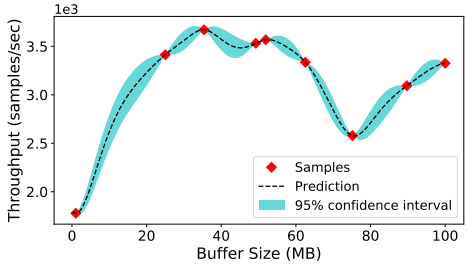


Fig. 3: Bayesian optimization example: 9 samples; tuning buffer size for training DenseNet-201.

Different from the previous works, DeAR pipelines some communication tasks with feed-forward computing tasks, which means that tensor fusion of any two layers may affect the granularity of feed-forward pipelining as shown in Fig. 2(c). Layer $L-1$ and layer $L-2$ are fused to be communicated once using OP_1 so that OP_2 of these two layers should also be invoked only once and it should be synchronized before the feed-forward computation of layer $L-2$. In this case, OP_2 of layer $L-1$ cannot be overlapped with the feed-forward computation of layer $L-2$. As a result, though DeAR reserves the property of tensor fusion, it is non-trivial to determine which layers should be fused to achieve minimal iteration time.

B. Bayesian optimization based tensor fusion

In DeAR, fusing the gradients of any two nearby layers has two drawbacks. 1) It requires to wait for the completion of the two layers' gradient computations to start the communication of reduce-scatter, which means the current ready layer cannot start communication immediately. 2) It reduces the opportunity of overlapping all-gather of one layer with the feed-forward computation of its previous layer. Thus, one should carefully choose the tensor fusion strategy such that the overall iteration time can be shortened. Due to the difficulty in formulating the tensor fusion problem with heuristic or optimal solutions, we choose to use Bayesian optimization (BO) [29,30], which attempts to find good parameters of an unknown objective function in as few number of trials as possible [16,25].

The target of BO used in DeAR is try to achieve maximum training performance (measured as the system throughput, i.e., the number of training samples that can be processed per second) during run-time in our system. We use $P(x)$ to denote the performance model of our system, which is unknown, and x is the buffer size which is an input parameter used for tensor fusion. Note that different x may generate different tensor fusion solutions. Specifically, nearby layers are put into one group if their total number of gradients does not exceed the

size x. Gradients in one group will have only one reduce-scatter operation during backpropagation, and one all-gather operation during feed-forward. We would like to update x dynamically such that $P(x)$ converges to a stable value.

BO is effective to find near-optimal tensor fusion solutions for three reasons. First, BO uses the Gaussian process regression in predicting the function value, so it has no constraints on the objective function format and only relies on the existing observations, i.e., $\hat{P}(x_1), \hat{P}(x_2), ..., \hat{P}(x_n)$. Second, BO usually needs a few number of trials to find good solutions, which only requires very small search costs. This is because BO suggests the next system configuration based on a well-defined acquisition function [31]. In this work, we use expected improvement (EI) acquisition function to pick the next point that can maximize the expected improvement over the current best result. Third, BO can tolerate uncertainty with quantitative confidence interval. For example, by tuning the EI hyper-parameter, we find BO can balance between exploitation and exploration during the search process, which is helpful to escape from a local optimum. In general, smaller EI hyper-parameter prefers exploitation (i.e., most points are around the peaks), while larger value prefers exploration (i.e., the points are more spread out across the whole range) [31]. In this problem, we set EI hyper-parameter as 0.1 to prefer buffer size exploration, e.g., from 1MB to 100MB (see Fig. 3).

To support BO during training, we first use a default buffer size $x_1 = 25$MB to initialize the tensor fusion configuration and measure the average system throughput (i.e., $\hat{P}(x_1)$) over multiple steps (e.g., 10 steps). Based on this measurement $\hat{P}(x_1)$, BO fits the performance function and suggests the next buffer size x_2, which can be used to generate a new tensor fusion solution. By repeating this process, BO can predict the performance accurately with enough samples, and find a good tensor fusion solution. For example, in Fig. 3, we use BO to find the buffer size for training DenseNet-201 [32] in DeAR. With only 9 samples, it returns a nearly optimal value at 35MB with a good confidence. In practice, tens of trials are enough to find a good solution for DeAR (see Figure 10), while the BO tuner developed for Horovod is much more costly, as it needs to search multiple system configurations including buffer size, cycle time, response cache, and hierarchical collective algorithms [16].

V. Implementation

We implement our prototype system, DeAR, based on PyTorch and NCCL. In the system, we wrap a communication library using C/C++ based on NCCL and expose APIs for high-level scripts in Python. The overview of our DeAR implementation is shown in Fig. 4. The blue components are new in DeAR, but they are totally transparent to end users. Users only need to change their code (normally with several new inserted lines) to use DeAR.

A. Workflow of DeAR

We implement our DeAR as a middle layer between user code and communication primitives. DeAR does not change the original DAG constructed in PyTorch. A *distributed optimizer* is implemented in DeAR to handle the gradient communications in hook functions provided by PyTorch APIs. Before communicating gradients through reduce-scatter or all-gather, gradients should be put in the *tensor fusion controller* which determines whether the pushed tensor should be copied to the buffer to be communicated together. When the buffer should be aggregated among all workers, DeAR invokes the wrapped NCCL APIs (e.g., ncclReduceScatter and ncclAllGather) in *communication package*.

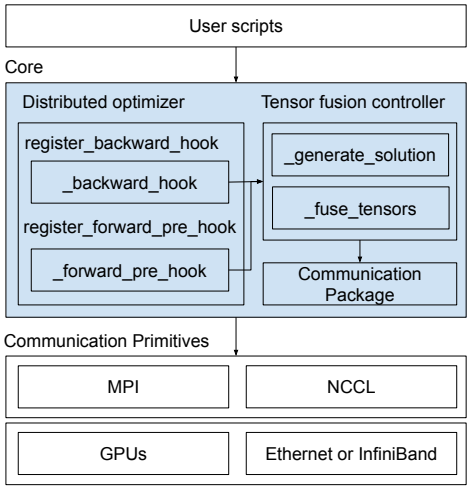


Fig. 4: Overview of our system (blue parts are new).

B. User usage

```
1  import dear # +++
2  dear.init() # +++
3  optim = torch.optim.SGD(model.parameters(), ...)
4  optim = dear.DistOptim(optim, model, ...) # +++
5  # Training
6  model.train()
7  for i in range(epochs):
8      for data, target in train_loader:
9          train_step(optim, data, target)
10         ...
11 # Validation
12 optim.synchronize() # +++
13 optim.step() # +++
14 model.eval()
15 validation()
```

Listing 1: Code example of using DeAR

To make our DeAR be easily integrated with existing user code, we design a distributed optimizer (DistOptim) that is exposed to users. Users only need to wrap their original PyTorch optimizer instances to our DistOptim and initialize a new instance of the optimizer as the sample code shown in Listing 1. The first two lines should be inserted to initialize the run-time of DeAR. Then line 4 is inserted after the standard optimizer instance and the training code remains unchanged. As DeAR pipelines the communication tasks of current iteration with the next iteration's feed-forward computing tasks, the communication tasks should be forced to synchronize

to update the model parameters (lines 12 and 13) before evaluating the model.

VI. Evaluation

A. Experimental settings

Testbeds. We conduct experiments on a 16-node dense-GPU cluster, which has 64 Nvidia GTX 2080Ti GPUs with four GPUs per node. The cluster is connected with both 10Gb/s Ethernet (10GbE) and 100Gb/s InfiniBand (100GbIB). Thus, we can choose two different network configurations to test the scalability of different algorithms. Each node has 512GiB RAM and the same software configurations. Specifically, each node is installed with Ubuntu18.04, CUDA-10.2, cuDNN-7.6, NCCL-2.10, OpenMPI-4.0, and PyTorch-1.8. We use NCCL APIs for all collective communications in our experiments.

DNN models. We choose two popular types of DNNs. They are image classification models, CNNs, on the ImageNet dataset [33], and NLP pre-training models BERT [34]. The detailed settings are shown in Table I. A training sample is an image with a resolution of $224 \times 224 \times 3$ for CNNs, and a sentence with a length of 64 words for BERTs.

TABLE I: DNN details for experiments. "BS" denotes the mini-batch size per GPU. "# Layers" represents the number of learnable layers. "# Tensors" and "# Param." denote the number of learnable parameter tensors and the number of elements (million) in these tensors, respectively.

Application	Model	BS	# Layers	# Tensors	# Param. (M)
Image Classification	ResNet-50 [35]	64	107	161	25.6
	DenseNet-201 [32]	32	402	604	20.0
	Inception-v4 [36]	64	299	449	42.7
NLP Pre-training	BERT-Base [34]	64	105	206	110.1
	BERT-Large [34]	32	201	398	336.2

Baselines. We compare our system with existing state-of-the-art systems including Horovod-0.21.3 [16], PyTorch-DDP [15] (at PyTorch-1.8), ByteScheduler[3], and MG-WFBP[4]. All the systems are based on the DL framework PyTorch.

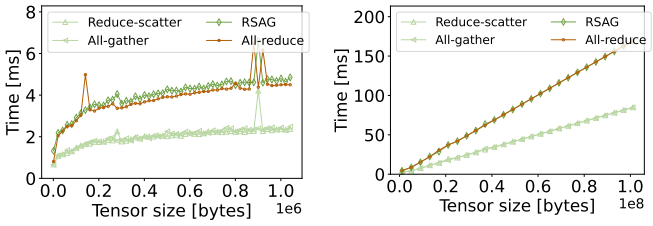
B. Verification of all-reduce breakdowns

To verify that the decoupling of all-reduce has almost zero overhead with different message sizes on dense-GPU clusters, we measure the elapsed-time of all-reduce and its decoupling methods (i.e., reduce-scatter, all-gather, and their combination). We run experiments using nccl-tests[5] on the 64-GPU cluster connected with 10GbE. The results are shown in Fig. 5, in which we can see that both reduce-scatter and all-gather take around half of the time of all-reduce with both small and large sizes messages. Thus, DeAR enables a finer-grained schedule for the decoupled communication tasks without introducing extra communication overheads.

[3]https://github.com/bytedance/byteps/tree/bytescheduler/bytescheduler (at GitHub commit 33fe89). Note that ByteScheduler cannot support PyTorch-1.8, so we configure PyTorch-1.4 when running ByteScheduler.

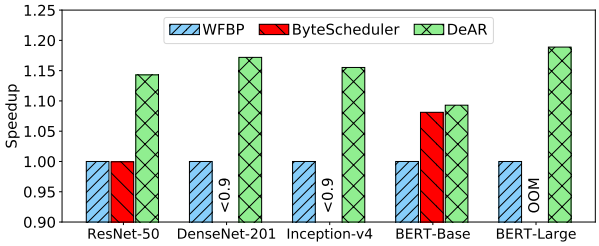
[4]https://github.com/HKBU-HPML/MG-WFBP (at GitHub commit 5b8ad5)

[5]https://github.com/NVIDIA/nccl-tests

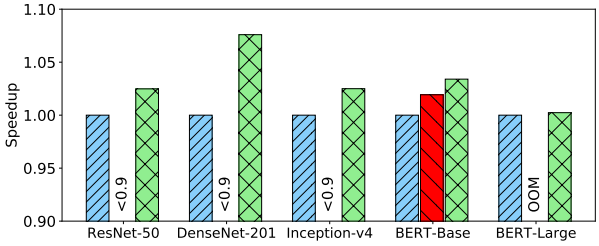


(a) Small size: (1K, 1M) (b) Large size: (1M, 100M)

Fig. 5: Performance comparison with different message aggregation methods. "RSAG" represents the all-reduce algorithm that is implemented with a reduce-scatter operation followed by an all-gather operation.



(a) 64 GPUs with 10GbE



(b) 64 GPUs with 100GbIB

Fig. 6: Speedups without tensor fusion. The performance of WFBP is used as the baseline. ByteScheduler runs out-of-memory (OOM) in BERT-Large.

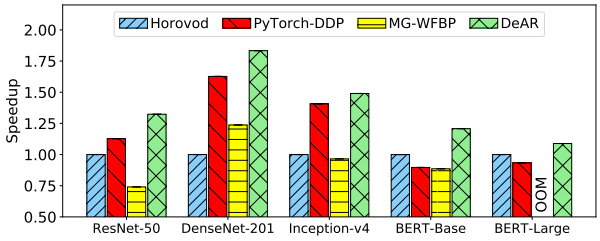
C. Speed comparison w/o tensor fusion

We first show the benefits of overlapping the communication tasks with feed-forward computing tasks in our DeAR without any tensor fusion techniques. We compare DeAR with existing scheduling algorithms without tensor fusion including WFBP [13] and ByteScheduler [25]. For a fair comparison with WFBP, we implement the all-reduce API with a reduce-scatter operation followed by an all-gather operation. The speedup results are shown in Fig. 6, using the performance of WFBP as the baseline.

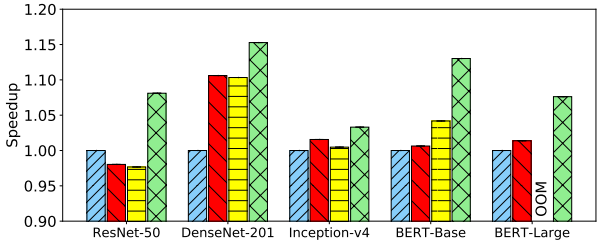
Compared with WFBP which only pipelines the communications with backpropagation computing tasks, our DeAR achieves 6%-19% improvement in all tested cases due to our fine-grained schedule where the communication tasks are pipelined with both feed-forward and backpropagation computing tasks.

ByteScheduler also pipelines the communications with both feed-forward and backpropagation computing tasks, but it

uses tensor partitioning and task re-ordering to achieve finer-grained scheduling of tasks. However, tensor partitioning and re-ordering require extensive extra communication overheads under the all-to-all architecture, ByteScheduler runs very slow in most cases especially on CNNs, thus its bars in Fig. 6 are very low (e.g., < 0.9). DeAR significantly outperforms ByteScheduler, especially on CNNs. In contrast, on BERT models which have much larger tensor sizes, the performance of ByteScheduler is relatively good since partitioning large tensor sizes does not introduce dramatic extra startup overheads.



(a) 64 GPUs with 10GbE



(b) 64 GPUs with 100GbIB

Fig. 7: Speedups with tensor fusion. The performance of Horovod is used as the baseline. MG-WFBP runs out-of-memory (OOM) in training BERT-Large.

D. Speed comparison w/ tensor fusion

Due to the high latency of the all-to-all collectives, tensor fusion has become a default feature to accelerate distributed training. Though DeAR decouples one all-reduce operation to two operations for better communication scheduling, we need to use tensor fusion for high latency collectives (§IV). We compare our DeAR with existing state-of-the-art algorithms including Horovod, PyTorch-DDP, and MG-WFBP [23], which are all equipped with tensor fusion techniques. For a fair comparison, we fix the buffer size of all algorithms with 25MB, except MG-WFBP. The results are shown in Fig. 7, where we use the performance of Horovod as the baseline.

On the 10GbE cluster. On the 64-GPU cluster with a relatively high-latency and low-bandwidth 10GbE network, we can see that DeAR with tensor fusion always outperforms all the other methods. Specifically, our DeAR achieves 6%-83% (an average of 36%) improvement over existing methods on the five tested models. DeAR achieves near-linear scaling efficiency on 64 GPUs in CNNs whose number of parameters is moderate (as shown in Table I). Though DeAR achieves significant improvement over other methods on BERT models, the scaling efficiency is still low due to the high communication-to-computation ratio. It typically requires some algorithmic-level optimizations like gradient compression [37]. We will leave it as our future work to introduce gradient compression techniques into our DeAR scheduling framework.

On the 100GbIB cluster. On the 64-GPU cluster with a low-latency and high-bandwidth 100GbIB network, the startup problem in distributed training is less significant. Even so, the end-to-end performance improvement of our DeAR over existing methods can be up to 15% (an average of 8%). In the 100GbIB network, the scalability of traditional methods like Horovod and PyTorch-DDP is close to linear scale in CNNs. For example, in the 64-GPU case of running ResNet-50, Horovod achieves 91% scaling efficiency leaving limited room for further improvement. Therefore, the improvement of DeAR in 100GbIB over other methods is less significant than that of 10GbE. Given the model size and network configurations, we discuss their maximum speedups in the next subsection.

E. Maximum speedups on 64-GPU clusters

Intuitively, on a P-GPU cluster, the maximum speedup over a single GPU should be P, i.e., linear scale. However, due to the communication constraint, the maximum speedup may be smaller than P. In DeAR, each training iteration time is composed of four parts: feed-forward computation (t_{ff}), backpropagation computation (t_{bp}), gradient communication of reduce-scatter (t_{rs}) and gradient communication of all-gather (t_{ag}). The all-reduce time is $t_{ar} = t_{rs}+t_{ag}$. Given a DL model with m gradient size and a cluster with P workers, the communication time should be larger than the time when the link bandwidth is fully utilized. For the ring-based all-reduce algorithm, $t_{ar} \geq 2m/B$ according to Eq. 5, where $B = 1/\beta$ is the minimum link bandwidth between any two workers. Thus, for any scheduling algorithms that pipeline communications with computations, the speedup of the overall throughput on the P-worker system over the single worker is limited by

$$S^{max} = \frac{P \times (t_{ff} + t_{bp})}{t_{ff} + t_{bp} + t_{ar} - \min\{t_{rs}, t_{bp}\} - \min\{t_{ag}, t_{ff}\}}, \tag{6}$$

where $\min\{t_{rs}, t_{bp}\}$ and $\min\{t_{ag}, t_{ff}\}$ are the overlapped time during backpropagation and feed-forward, respectively. Optimally, either computations or communications are fully hidden. According to the link bandwidth of 10GbE and 100GbIB and the model size shown in Table I, we can compare the achieved speedups of DeAR on the 64-GPU cluster over a single GPU with the maximum speedups as shown in Table. II.

It is seen that given a high communication-to-computation ratio, the linear scaling efficiency may not be reachable. For example, on the 10GbE cluster, theoretically, running BERT models on 64 GPUs can only achieve less than 25.5 times speedup over a single GPU. In our DeAR which has two level of pipelining between communications and computations, it achieves an average of 93.6% and 83.9% of the theoretical

TABLE II: Comparison between the real speedup (S) of DeAR on 64-GPU clusters over single GPU and the theoretical maximal speedup (S^{max}).

		Model				
		ResNet-50	DenseNet-201	Inception-v4	BERT-Base	BERT-Large
10-GbE	S^{max}	61.6	64	59.8	25.5	12.1
	S	61.1	52.8	56.5	23.9	11.8
	$\frac{S}{S^{max}}$	99.2%	82.5%	94.5%	93.9%	98.0%
100-GbIB	S^{max}	64	64	64	64	51.8
	S	61.6	54.0	57.2	49.6	37.5
	$\frac{S}{S^{max}}$	96.2%	84.4%	89.4%	77.5%	72.3%

optimal speedup on 10GbE and 100GbIB interconnects, respectively.

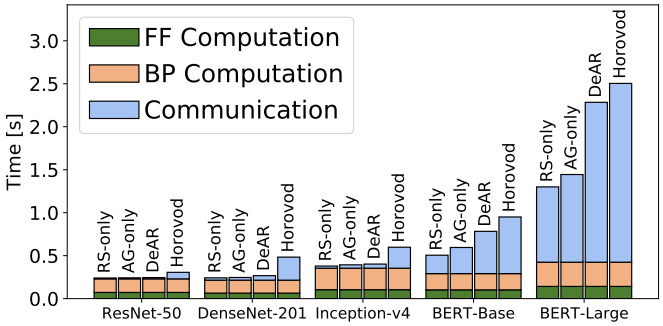


Fig. 8: Time breakdowns. The communication time excludes the part hidden by computations.

F. Time breakdowns

To understand the improvement of DeAR over existing methods clearer, we break down the iteration time into three parts: feed-forward (FF) computation time, backpropagation (BP) computation time, and gradient aggregation communication time (under the 10GbE network configuration) as shown in Fig. 8. The blue bars are the non-overlapped communication time in one iteration, which means the hidden time by computations is excluded. As both DeAR and Horovod use the same back-end of PyTorch, the FF and BP computation times are the same on the same model. In DeAR, there are two parts of communication, reduce-scatter and all-gather. We use RS-only to indicate that DeAR excludes the time of all-gather, and AG-only to indicate that DeAR excludes the time of reduce-scatter in Fig. 8.

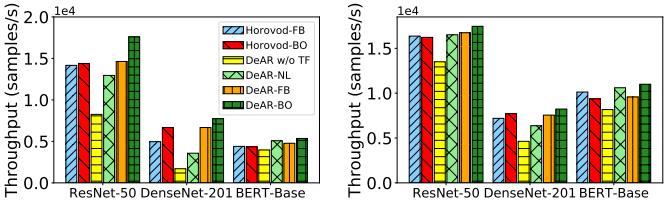
RS-only vs. AG-only. In the decoupling of all-reduce, reduce-scatter and all-gather operations have the same message size for communication and they have the same communication complexity as shown in Eq. 3 and Eq. 4. In other words, the communication time of reduce-scatter and all-gather operations should be the same without considering the overlapping with computations. However, as shown in Fig. 8, RS-only has a less communication overhead than AG-only. The reason is that the reduce-scatter operations are overlapped with backpropagation computing tasks which typically take two times slower than feed-forward computing tasks [18]. Thus, reduce-scatter has more communication tasks to be overlapped with computing tasks than all-gather.

DeAR vs. Horovod. In the results on the breakdown of the iteration time, we can see that DeAR has a smaller non-overlapped communication time than Horovod. Though our DeAR has a similar pipelining with Horovod during backpropagation, DeAR has the opportunity to further pipeline the communications with feed-forward computations, which contributes to the performance improvement.

G. Studies of tensor fusion

Tensor fusion is an important technique to improve performance as shown in Fig. 9. The tensor fusion version of DeAR with BO (DeAR-BO) achieves 1.35×-4.54× and 1.29×-1.78× improvement over the version w/o tensor fusion (DeAR w/o TF) under 10GbE and 100GbIB interconnects, respectively.

1) Performance comparison between w/ BO and w/o BO



(a) On the 10GbE cluster (b) On the 100GbIB cluster

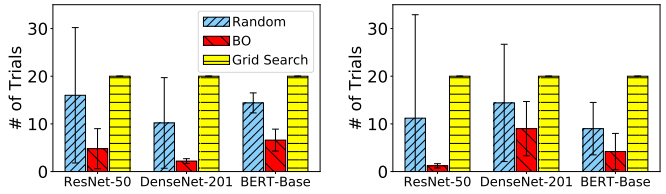
Fig. 9: Speed improvements with dynamic tensor fusion.

Determining which tensors should be fused is very important in affecting the training performance. We compare our BO-based tensor fusion method (DeAR-BO) with two naive tensor fusion methods, a fixed number of four nearby layers (DeAR-NL) and a fixed buffer size (5MB) threshold (DeAR-FB), and BO-based Horovod. The results are shown in Fig. 9.

Horovod-BO vs. Horovod-FB. Horovod with BO (Horovod-BO) achieves only slight improvement in ResNet-50 and DenseNet-201 over Horovod with a fixed buffer (Horovod-FB) size (64MB by default), while Horovod-BO has no improvement in BERT-Base over Horovod-FB. The results indicate that Horovod-FB may be a good solution in tensor fusion for WFBP and tuning the buffer size does not help improve the performance.

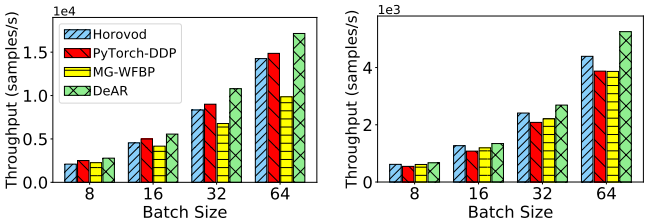
DeAR-NL vs. DeAR-FB. By merging a fixed number of consecutive layers, DeAR-NL is normally worse than Horovod-FB or DeAR-FB in CNNs as CNNs have a very imbalanced number of parameters in different layers. For the model (BERT-Base) that has a very balanced distribution of parameters in different layers, DeAR-NL performs better than Horovod-FB and DeAR-FB.

DeAR-FB vs. Horovod-FB. With the opportunity of overlapping communications with both feed-forward and backpropagation computing tasks, DeAR-FB normally outperforms Horovod-FB. Since DeAR requires to pipeline tasks during feed-forward, the configured buffer size is more sensitive to the time performance than Horovod-FB. Thus, DeAR-FB only achieves a marginal improvement over Horovod-FB.



(a) On the 10GbE cluster (b) On the 100GbIB cluster

Fig. 10: Tuning cost of different search algorithms. Error bars show standard deviation.



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

Fig. 11: Speed comparison with different batch sizes.

DeAR-BO vs. others. Using BO in DeAR, the buffer size can be well adjusted during run-time. Hence, DeAR-BO can improve the training speed over DeAR-FB, and it achieves the best performance among all the evaluated methods. Specifically, DeAR-BO is around 22%-56% and 7%-14% faster than Horovod-FB on 10GbE and 100GbIB 64-GPU clusters, respectively.

2) Search cost comparison between BO and other methods

In DeAR, we use BO to find a good buffer size. Actually, one can also use random search or grid search to find the solution. However, both random search and grid search take a much larger number of iterations to find a good solution. For example, as shown in Fig.10, in training different models, BO takes several trials to find a stable solution while random and grid search take tens of trials. In DNN training, one cannot consume too much number of iterations to tune their time performance related parameters, otherwise it may result in a longer end-to-end training time than the naive methods. The average cost of BO is 0.207 seconds per trial over 20 trials.

H. Performance with different batch sizes

The local mini-batch size has a direct impact on the feed-forward and backpropagation computation time, thus training with different batch sizes has different communication-to-computation ratios on the same model, which would affect the opportunity for overlapping. We compare our DeAR with the existing methods on the 10GbE cluster using ResNet-50 and BERT-Base under different batch sizes as shown in Fig. 11. The batch size is the local mini-batch size for each GPU running on the 64-GPU clusters. Smaller batch size indicates shorter computation time while the communication time remains unchanged (if not overlapped). The results show that our DeAR is robust to the mini-batch size and it outperforms all other methods in all tested cases.

I. Potential improvement on larger-scale clusters

Due to the hardware limit, we are unable to perform experiments on larger-scale GPU clusters to verify the performance of DeAR. We provide some discussions here. According to Fig. 7, we can see that the improvement of our DeAR over existing methods on 100GbIB (an average of 8%) is much smaller than on 10GbE (an average of 36%). In our experiment environments, the two clusters are with the same GPU hardware, which means the FF and BP computation time remain unchanged while the network speed of 100GbIB is 10 times faster than 10GbE. That is, existing methods should be better scaling efficiency with a higher network speed, which makes the optimization room be smaller. For example, in ResNet-50, Horovod achieves 58.2× speedup on 64 GPUs with 100GbIB over the single GPU while the maximum speedup is only 64× (Table II). Therefore, any optimization cannot achieve improvement larger than 10% over Horovod. We argue that with increasing size of the cluster and more powerful GPUs, DeAR could have a higher improvement over Horovod and PyTorch-DDP as the communication-to-computation ratio becomes larger.

Formally, the theoretical optimal iteration time for DeAR and baseline algorithms with perfect overlapping is given by

$$t_{DeAR} = \max\{t_{ff}, t_{ag}\} + \max\{t_{bp}, t_{rs}\}, \tag{7}$$

$$t_{baseline} = t_{ff} + \max\{t_{bp}, t_{ar}\}, \tag{8}$$

where t_{ff} and t_{bp} are feed-forward and back-propagation computation time, t_{rs}, t_{ag}, and t_{ar} are reduce-scatter, all-gather, and all-reduce communication time, respectively. Assume that $t_{ar} = 2t_{rs} = 2t_{ag}$ and $t_{bp} = 2t_{ff}$, we can derive that

$$t_{baseline} - t_{DeAR} = \begin{cases} 0, & \text{if } t_{ag} \le t_{ff}, \\ t_{ag} - t_{ff}, & \text{if } t_{ff} < t_{ag} \le 2t_{ff}, \\ t_{ff}, & \text{otherwise.} \end{cases} \tag{9}$$

This implies that, if we implement DeAR properly to overlap communications with forward and backward computations, DeAR can always outperform baseline algorithms such as Horovod and PyTorch-DDP. As the communication-to-computation ratio becomes larger, the saved iteration time can be at most one feed-forward computation cost of t_{ff}.

VII. Related Work

There are many studies in addressing the communication problem of distributed training, like asynchronous training [38, 39] and gradient compression [40]–[43]. A more comprehensive introduction can be found in recent survey papers [44,45]. Here we highlight some very related work from the systems perspective in S-SGD.

A. Efficient all-reduce design

As the all-reduce collective is very ubiquitous in distributed deep learning, there are many works providing efficient algorithms for different cluster configurations [2,7,8,14,37,46]–[50]. For example, the double-binary trees algorithm [46] has been integrated in NCCL to scale-out on extremely large-scale GPU clusters. Some particular all-reduce algorithms are also designed for different cluster topology, like [51] on Torus networks, [8,48] on NVLink-based GPU servers, [37,47] on public cloud clusters, and [50] on heterogeneous GPU clusters. These designs are orthogonal to our DeAR as long as these algorithms can be decoupled into two operations without introducing extra overheads. For example, one can decompose the double-binary tree-based all-reduce [46] into tree-based reduce and tree-based broadcast, and decompose the hierarchical ring-based all-reduce [51] into intra-node and inter-node reduce-scatter and all-gather. We leave decoupling more all-reduce algorithms as our future work, and the decoupling configuration can be automatically tuned using BO.

B. Communication scheduling

Due to the layer-wise structure of DNN models and even tensor-wise in modern deep learning frameworks like PyTorch and TensorFlow, the computing and communication tasks are generally organized as a directed acyclic graph (DAG). Thus, the tasks without any dependency can be executed concurrently, making it possible to hide some communication costs by overlapping them with computing tasks. The wait-free backpropagation (WFPB) algorithm [13,14] was the early scheduling method that pipelines the communication tasks of gradient aggregation with gradient calculation during back-propagation. Then some tensor fusion techniques (e.g., [17,23, 24,52,53]) were further proposed to address the high latency problem in WFBP with all-reduce. As the communication of some large tensors may block the execution of higher-priority tensors, ByteScheduler [25] proposes tensor partitioning in priority scheduling to provide a finer-grained schedule of overlapping between computing and communication tasks. ZeRO [54] decoupled all-reduce like DeAR, but it was to shard parameters to optimize memory rather than optimizing communication efficiency. To shard parameters, ZeRO requires one all-gather for each forward pass and one extra all-gather for each backward pass, which unfortunately has increased the total communication overheads compared with DeAR. In the recent PyTorch v1.13 release, FullyShardedDataParallel[6] has combined the ideas of ZeRO's parameter sharding and DeAR's FeedPipe to alleviate the memory and communication overheads, respectively, but it does not consider dynamic tensor fusion to explore the optimal training performance.

VIII. Conclusion

In this work, we proposed a novel communication scheduling algorithm named DeAR with fine-grained all-reduce pipelining. In DeAR, we first decoupled the all-reduce primitive into two continuous communication operations without introducing any extra communication overhead to enable a fine-grained schedule of computations and communications. Then we proposed to pipeline the backpropagation and feed-forward computing tasks with the first operations and second operations of the decoupled all-reduce primitives, respectively. To integrate the effective tensor fusion technique, we proposed a practical tensor fusion method using Bayesian optimization in DeAR to further reduce the communication time without considering the DNN model and network configuration. Extensive experiments were conducted on a 64-GPU cluster connected with two types of networks (10Gb/s Ethernet and 100Gb/s InfiniBand) on different applications including CNNs and BERTs. Experimental results show that DeAR achieves up to 83% improvement over existing state-of-the-art methods.

[6]https://pytorch.org/docs/stable/fsdp.html

Acknowledgments

The research was supported in part by a RGC RIF grant under the contract R6021-20, RGC GRF grants under the contracts 16209120, 16200221, 16207922, 16213120, the National Natural Science Foundation of China (NSFC) (Grant No. 62272122), and the National Natural Science Foundation of China (NSFC) (Grant No. 62002240).

References

[1] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang *et al.*, "Large scale distributed deep networks," in *Proc. of NeurIPS*, 2012, pp. 1223–1231.

[2] X. Jia, S. Song, S. Shi, W. He, Y. Wang, H. Rong, F. Zhou, L. Xie, Z. Guo, Y. Yang, L. Yu, T. Chen, G. Hu, and X. Chu, "Highly scalable deep learning training system with mixed-precision: Training ImageNet in four minutes," in *Proc. of Workshop on Systems for ML and Open Source Software, collocated with NeurIPS 2018*, 2018.

[3] Y. You, J. Li, S. Reddi, J. Hseu, S. Kumar, S. Bhojanapalli, X. Song, J. Demmel, K. Keutzer, and C.-J. Hsieh, "Large batch optimization for deep learning: Training bert in 76 minutes," in *International Conference on Learning Representations*, 2020.

[4] 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 preprint arXiv:1706.02677*, 2017.

[5] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "GPipe: Efficient training of giant neural networks using pipeline parallelism," *Proc. of NeurIPS*, vol. 32, 2019.

[6] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro *et al.*, "Efficient large-scale language model training on GPU clusters using megatron-LM," in *Proc. of SC*, 2021, pp. 1–15.

[7] A. A. Awan, K. Hamidouche, A. Venkatesh, and D. K. Panda, "Efficient large message broadcast using NCCL and CUDA-aware MPI for deep learning," in *Proc. of The 23rd European MPI Users' Group Meeting*, 2016, pp. 15–22.

[8] M. Cho, U. Finkler, and D. Kung, "Blueconnect: Novel hierarchical all-reduce on multi-tired network for deep learning," in *Proceedings of the Conference on Systems and Machine Learning (SysML)*, 2019.

[9] C.-H. Chu, P. Kousha, A. A. Awan, K. S. Khorassani, H. Subramoni, and D. K. Panda, "NV-group: link-efficient reduction for distributed deep learning on modern dense GPU systems," in *Proceedings of the 34th ACM International Conference on Supercomputing*, 2020, pp. 1–12.

[10] S. Shi, Z. Tang, X. Chu, C. Liu, W. Wang, and B. Li, "A quantitative survey of communication optimizations in distributed deep learning," *IEEE Network*, vol. 35, no. 3, pp. 230–237, 2020.

[11] P. Xu, S. Shi, and X. Chu, "Performance evaluation of deep learning tools in docker containers," in *2017 3rd International Conference on Big Data Computing and Communications (BIGCOM)*. IEEE, 2017, pp. 395–403.

[12] S. Shi, W. Qiang, and X. Chu, "Performance modeling and evaluation of distributed deep learning frameworks on GPUs," in *Proc. of The 4th International Conference on Big Data Intelligence and Computing*. IEEE, 2018.

[13] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing, "Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 181–193.

[14] A. A. Awan, K. Hamidouche, J. M. Hashmi, and D. K. Panda, "S-caffe: Co-designing MPI runtimes and Caffe for scalable deep learning on modern GPU clusters," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2017, pp. 193–205.

[15] S. Li, Y. Zhao, R. Varma *et al.*, "Pytorch distributed: Experiences on accelerating data parallel training," *Proc. of VLDB*, vol. 13, no. 12.

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

[17] J. Romero, J. Yin, N. Laanait, B. Xie, M. T. Young, S. Treichler, V. Starchenko, A. Borisevich, A. Sergeev, and M. Matheson, "Accelerating collective communication in data parallel training across deep learning frameworks," in *Proc. of NSDI*, 2022, pp. 1027–1040.

[18] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[19] M. Barnett, L. Shuler, R. van de Geijn, S. Gupta, D. Payne, and J. Watts, "Interprocessor collective communication library (intercom)," in *Proceedings of IEEE Scalable High Performance Computing Conference*, 1994, pp. 357–364.

[20] R. Rabenseifner, "Optimization of collective reduction operations," in *International Conference on Computational Science*. Springer, 2004, pp. 1–9.

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

[22] T. Hoefler, W. Gropp, R. Thakur, and J. L. Träff, "Toward performance models of MPI implementations for understanding application scaling issues," in *European MPI Users' Group Meeting*. Springer, 2010, pp. 21–30.

[23] S. Shi, X. Chu, and B. Li, "MG-WFBP: Efficient data communication for distributed synchronous SGD algorithms," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 172–180.

[24] ——, "MG-WFBP: Merging gradients wisely for efficient communication in distributed deep learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 8, pp. 1903–1917, 2021.

[25] 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. of SOSP*, 2019, pp. 16–29.

[26] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server." in *OSDI*, vol. 14, 2014, pp. 583–598.

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

[28] R. Thakur and W. D. Gropp, "Improving the performance of collective operations in MPICH," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 2003, pp. 257–267.

[29] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," *Proc. of NeurIPS*, vol. 25, 2012.

[30] F. Nogueira, "Bayesian Optimization: Open source constrained global optimization tool for Python," 2014–. [Online]. Available: https://github.com/fmfn/BayesianOptimization

[31] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *NSDI*, 2017.

[32] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.

[33] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. of CVPR*, 2009, pp. 248–255.

[34] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 2019, pp. 4171–4186.

[35] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. of CVPR*, 2016, pp. 770–778.

[36] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Proc. of The 31st AAAI*, 2017.

[37] S. Shi, X. Zhou, S. Song, X. Wang, Z. Zhu, X. Huang, X. Jiang, F. Zhou, Z. Guo, L. Xie *et al.*, "Towards scalable distributed training of deep learning on public cloud clusters," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 401–412, 2021.

[38] S. Zheng, Q. Meng, T. Wang, W. Chen, N. Yu, Z.-M. Ma, and T.-Y. Liu, "Asynchronous stochastic gradient descent with delay compensation," in *International Conference on Machine Learning*, 2017, pp. 4120–4129.

[39] X. Lian, Y. Huang, Y. Li, and J. Liu, "Asynchronous parallel stochastic gradient for nonconvex optimization," in *Proc. of NeurIPS*, 2015, pp. 2737–2745.

[40] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "QSGD: Communication-efficient SGD via gradient quantization and encoding," in *Proc. of NeurIPS*, 2017, pp. 1709–1720.

[41] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep gradient compression: Reducing the communication bandwidth for distributed training," in *International Conference on Learning Representations*, 2018.

[42] S. Shi, K. Zhao, Q. Wang, Z. Tang, and X. Chu, "A convergence analysis of distributed SGD with communication-efficient gradient sparsification." in *IJCAI*, 2019, pp. 3411–3417.

[43] D. Basu, D. Data, C. Karakus, and S. Diggavi, "Qsparse-local-SGD: Distributed SGD with quantization, sparsification and local computations," in *Proc. of NeurIPS*, 2019, pp. 14 695–14 706.

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

[45] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, pp. 1–43, 2019.

[46] P. Sanders, J. Speck, and J. L. Träff, "Two-tree algorithms for full bandwidth broadcast, reduction and scan," *Parallel Computing*, vol. 35, no. 12, pp. 581–594, 2009.

[47] L. Luo, P. West, J. Nelson, A. Krishnamurthy, and L. Ceze, "PLink: Discovering and exploiting locality for accelerated distributed training on the public cloud," in *Proceedings of Machine Learning and Systems 2020*, 2020, pp. 82–97.

[48] G. Wang, S. Venkataraman, A. Phanishayee, N. Devanur, J. Thelin, and I. Stoica, "Blink: Fast and generic collectives for distributed ML," in *Proceedings of Machine Learning and Systems 2020*, 2020, pp. 172–186.

[49] J. Dong, Z. Cao, T. Zhang, J. Ye, S. Wang, F. Feng, L. Zhao, X. Liu, L. Song, L. Peng *et al.*, "EFLOPS: Algorithm and system co-design for a high performance distributed training platform," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 610–622.

[50] X. Miao, X. Nie, Y. Shao, Z. Yang, J. Jiang, L. Ma, and B. Cui, "Heterogeneity-aware distributed machine learning training via partial reduce," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 2262–2270.

[51] H. Mikami, H. Suganuma, Y. Tanaka, Y. Kageyama *et al.*, "Massively distributed sgd: Imagenet/resnet-50 training in a flash," *arXiv preprint arXiv:1811.05233*, 2018.

[52] Y. You, A. Buluç, and J. Demmel, "Scaling deep learning on GPU and knights landing clusters," in *Proc. of SC*, 2017, pp. 1–12.

[53] S. Shi, X. Chu, and B. Li, "Exploiting simultaneous communications to accelerate data parallel distributed deep learning," in *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 2021, pp. 1–10.

[54] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models," in *Proc. of SC*. IEEE, 2020, pp. 1–16.