Preemptive All-reduce Scheduling for Expediting Distributed DNN Training

Yixin Bao*, Yanghua Peng*, Yangrui Chen*, Chuan Wu*
*Department of Computer Science, The University of Hong Kong, Email: {yxbao,yhpeng,yrchen,cwu}@cs.hku.hk

***Abstract*—Data-parallel training is widely used for scaling DNN training over large datasets, using the parameter server or all-reduce architecture. Communication scheduling has been promising to accelerate distributed DNN training, which aims to overlap communication with computation by scheduling the order of communication operations. We identify two limitations of previous communication scheduling work. First, layer-wise computation graph has been a common assumption, while modern machine learning frameworks (*e.g.*, TensorFlow) use a sophisticated directed acyclic graph (DAG) representation as the execution model. Second, the default sizes of tensors are often less than optimal for transmission scheduling and bandwidth utilization. We propose *PACE*, a communication scheduler that preemptively schedules (potentially fused) all-reduce tensors based on the DAG of DNN training, guaranteeing maximal overlapping of communication with computation and high bandwidth utilization. The scheduler contains two integrated modules: given a DAG, we identify the best tensor-preemptive communication schedule that minimizes the training time; exploiting the optimal communication scheduling as an oracle, a dynamic programming approach is developed for generating a good DAG, which merges small communication tensors for efficient bandwidth utilization. Experiments in a GPU testbed show that *PACE* accelerates training with representative system configurations, achieving up to 36% speed-up compared with state-of-the-art solutions.**

I. Introduction

Deep learning has developed significantly in recent years on a variety of applications as in computer vision and natural language processing. Training of large deep neural networks (DNNs) on increasing volumes of data is very time-consuming. Data parallelism with synchronous parameter update [1] [2] is a popular method for distributed DNN training with better convergence guarantee, where in each training iteration, each worker has a full copy of model parameters, trains a subset of the input data, exchanges parameter updates (gradients) with each other, and then updates model parameters. As workers exchange gradients frequently, communication often occupies a significant portion in overall training time (which can be as high as 90%) [3] [4] [5].

Different approaches have been proposed for communication acceleration in distributed DNN training. One can reduce communication traffic using techniques such as gradient quantization [4], mixed-precision training [6], or can deploy higher bandwidth networks or RDMA [7]. They target a common goal – accelerating individual communication operations.

A new direction to accelerate distributed training is to schedule communication such that it is best overlapped with computation. In each training iteration, gradients can be sent immediately after they are calculated, without waiting for the completion of the entire backward process. This is known as wait-free backward propagation (WFBP) [3] and is supported in most distributed training frameworks, *e.g.*, TensorFlow [8], PyTorch [9], MXNet [10] and B-Caffe [1]. First-In-First-Out (FIFO) is the default policy for scheduling communication tensors (*i.e.*, gradients to be transmitted) according to their generation order, in existing machine learning (ML) systems such as MXNet and TensorFlow. After gradient synchronization among workers, updated parameters are not needed simultaneously for activation calculation in the next iteration, but according to the sequence of computation operations consuming them [5] [11]. In this case, FIFO tensor communication scheduling is less than optimal: we can hide the communication time better by adjusting the order of tensor communications, to overlap them partly with activation calculation (of the next iteration), without affecting training results (Sec. II-C) [12]. Unfortunately, it is not easy to achieve such communication scheduling in modern ML frameworks [8] [9] due to two main reasons (Sec. II).

First, while a DNN is often defined as a layer-by-layer structure by users, the computation graph of the DNN is not a layered structure. State-of-the-art ML frameworks represent the training execution model as a sophisticated Directed Acyclic Graph (DAG), where the nodes represent operators of computation or communication, and the edges represent dependencies among operators. Previous communication scheduling solutions (*e.g.*, [1] [12] [13]) are restricted to layered computation graphs and not easily applicable to DAG representation used by ML engines. In a DAG, finding an optimal communication schedule efficiently is a non-trivial challenge.

Second, the default sizes of gradient tensors are often not ideal for scheduling and efficient bandwidth utilization. On the one hand, the head-of-line blocking issue may occur when a communication operator with a large tensor is scheduled: when another tensor with a higher priority (*e.g.*, needed sooner for next iteration) arrives later, it has to wait for the completion of the large tensor before its communication. On the other hand, small communication tensors (*e.g.*, less than 1MB) can not fully utilize network bandwidth, due to the relatively larger portion of overhead in its communication (*e.g.*, time spent on negotiation/synchronization among workers). The overhead is non-trivial especially for all-reduce operations, due to the need of strict synchronization among workers [1].

To address these issues, we design *PACE*, a communication

This work was supported in part by grants from Hong Kong RGC under the contracts HKU 17204619, 17225516, C5026-18G (CRF).

scheduler that preemptively schedules (potentially fused) all-reduce communication tensors based on the DAG of DNN training model, guaranteeing maximal overlapping of communication with computation and efficient bandwidth utilization. We focus on all-reduce gradient exchange paradigm in synchronous data-parallel training, where workers exchange gradients directly among each other without involving parameter servers. All-reduce is one widely adopted tensor communication approach for distributed DNN training, supported by various ML frameworks (*e.g.*, TensorFlow, PyTorch, PaddlePaddle [14], Horovod [15]), and demonstrating better scalability than the parameter server architecture [15].

PACE aims to identify the *best schedule* and *granularity* of tensor communication to maximally overlap communication with computation, in order to minimize the execution time of the training DAG. It contains two integrated modules. First, we devise a theoretically optimal approach for preemptive scheduling of all-reduce communication operators according to the given (fused) DAG. The preemption enables maximal communication-computation overlap by preventing head-of-line blocking of large communication tensors. Next, exploiting the optimal communication scheduling as an oracle, we propose a dynamic programming approach to generate a good DAG, by fusing small communication tensors to mitigate communication overhead incurred by a large number of small tensors and to achieve efficient bandwidth utilization. *PACE* expedites distributed DNN training without requiring changes in the DNN model or user code. Specifically, we make the following contributions in developing *PACE*.

▷ We formulate a preemptive communication scheduling problem for a general computation-communication dependency graph for all-reduce based DNN training, and develop a theoretically *optimal* scheduling solution (Sec. IV-B). The preemptive scheduling achieves variable tensor size partition for maximal overlap of computation and communication.

▷ We design a dynamic programming approach to fuse small communication tensors and generate an efficient computation-communication dependency graph to use (Sec. IV). The communication scheduling module is integrated as an oracle, to provide an execution-time minimization tensor schedule on each given candidate graph, for efficiently identifying the near-optimal graph. *PACE* produces the (fused) DAG and optimal communication schedule on it in polynomial time.

▷ We have implemented *PACE* scheduler on Horovod [15], using MXNet as the training framework. We evaluated *PACE* using 4 representative DNNs on a GPU cluster under different system settings (Sec. VI). Experiment results show that *PACE* accelerates training speed by up to 36% compared to WFBP, and up to 32% compared to Horovod.

II. Background and Motivation

A. Distributed DNN Training

DNN training. In deep learning, a DNN model is trained to minimize a loss function with a large dataset, which is equally divided into *mini-batches*. In each *iteration*, we pass a mini-batch through the DNN to obtain the loss, calculate the gradients, and then update parameters based on an optimization algorithm, *e.g.*, Stochastic Gradient Descent (SGD) [16]. The iterative training is repeated until the DNN converges.

All-reduce. In each iteration of distributed DNN training, gradient exchange among workers is typically through parameter server(s) or the all-reduce collective. All-reduce is a collective operation that sums (or averages) the gradients from all workers and disperses the aggregated (or averaged) gradients to them [17]. This operation can be implemented using various algorithms [1] [17]. Among them, ring all-reduce algorithm is most widely adopted in ML frameworks (*e.g.*, Horovod, PaddlePaddle), since it is proved to be bandwidth-optimal, assuming that the all-reduce tensor (*i.e.*, gradients to be aggregated or averaged by the all-reduce operation) is large enough [17]. Fig. 1 illustrates a ring all-reduce process among 3 workers, for summing gradients generated by the dReLu operator (AR_1). The workers form a ring topology. Each tensor is segmented evenly according to the number of workers, and the tensor chunks are reduced (*i.e.*, aggregated) by walking the ring from different starting points.

B. Computation-communication Dependency DAG

Dependency DAG. State-of-the-art ML frameworks [8] [9] use a dataflow graph to represent the training steps that a worker carries out in each training iteration, in terms of the dependencies among operators. The dataflow graph is typically a sophisticated Directed Acyclic Graph (DAG), where the nodes represent computation operators or communication operators, and the directed edges represent their dependencies. The ML engine executes an operator once it is ready, *i.e.*, all operators that it depends on have been finished.

Fig. 1 illustrates a contrived DAG in MXNet with 9 operators. Blue circles represent computation operators, and green shapes indicate communication operators. For example, Add depends on Mul and AR_2, AR_2 depends on $dAdd$, and $dAdd$ depends on $dReLu$. When executing the graph, the framework engine has no idea of coarse DNN layers, but only concepts of such fine-grained computation operators (*e.g.*, MatMul, BiasAdd, ReLu), communication operators (*e.g.*, all-reduce) and their dependencies.

C. Opportunities

Preemptive scheduling. By default, ML framework engines (such as TensorFlow, MXNet and PyTorch) execute communication operations in a FIFO order, because the underlying communication library (*e.g.*, ZMQ [18] in MXNet) is typically based on FIFO queues. As in Fig. 2 (corresponding to the DAG in Fig. 1), AR_1 is ready for execution first and then gets executed first; then AR_2 is executed before AR_3 (once AR_1 is finished, both AR_2 and AR_3 are ready). However, if we schedule AR_3 ahead of AR_2, Mul (which depends on AR_3) can start earlier, and its computation is overlapped with AR_2, resulting in reduced DAG execution time [5], [12].

In a DAG, the sizes of communication tensors can vary significantly (*e.g.*, the smallest tensor is 256B, and the largest tensor is over 400MB for VGG16 [19]). A very large tensor,

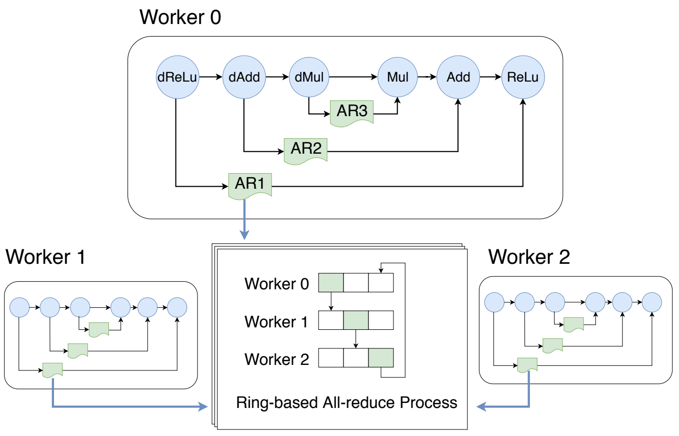


Fig. 1: An example DAG in MXNet and the all-reduce process.

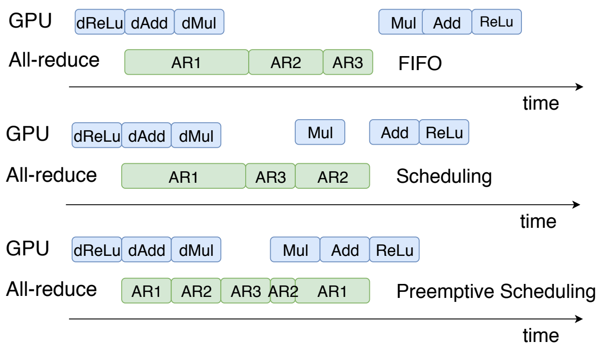


Fig. 2: Example scheduling strategies.

once started all-reduce, would block other tensors arriving later than the large tensor, even if they need to be finished earlier. Fig. 2 shows such a case: when AR_2 is ready, AR_1 has not been finished and hence blocks AR_2.

Therefore, only scheduling the transmission order of tensors is not sufficient. A better strategy is to allow execution preemption of the respective communication operator and resume it at a later time. Since all-reduce is a primitive operation, to achieve preemption of an all-reduce communication operator, we can segment it into multiple operators, each responsible for all-reduce of one partition of the original gradient tensor. The smaller operators can be scheduled at different times to allow newly-arrived all-reduce operators to be executed earlier. For example, AR_1 is partitioned into two smaller all-reduce operators with *uneven* tensor partition, so that the completion time of the first operator equals the ready time of AR_2. AR_2 is partitioned too to allow AR_3 to be executed once it is ready. After AR_3 is completed, the second partition of AR_2 will be scheduled, followed by that of AR_1. Compared with FIFO, this preemptive schedule achieves 25% reduction of DAG execution time. Finding the optimal preemption/partitioning points of all-reduce operators in a DAG with large tensors is a challenging task. We formulate the preemptive scheduling problem and devise an optimal solution in Sec. IV.

Tensor fusion. When the size of a gradient tensor is small (*e.g.*, less than 1MB), all-reduce can not fully utilize the network bandwidth: the communication overhead (*e.g.*, ACK time for message transmission, negotiation time among all workers before performing all-reduce on a tensor [15]) is non-negligible as compared to the short transmission time. Fig. 3 shows the average measured execution time of ring all-reduce operators with tensors of different sizes, obtained by running all-reduce benchmark in our GPU cluster (see Sec. VI for hardware details). Note that both x-axis and y-axis are in log scale. When the tensor size is small (less than 1MB), the all-reduce time does not decrease linearly with the decrease of tensor size. Fig. 4 shows that over 50% gradient tensors in ResNet50 [20] and Transformer [21] are less than 1MB in size. For DNNs with these many tiny tensors, the overhead in ring all-reduce communication can be significant.

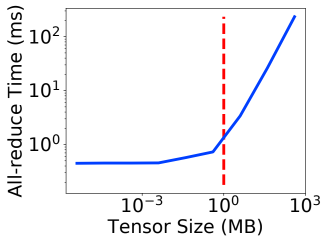


Fig. 3: Ring all-reduce time with different tensor sizes.

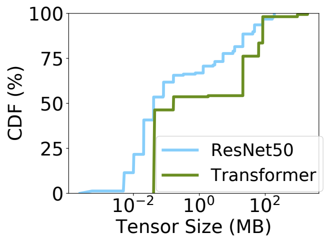


Fig. 4: Cumulative distribution function (CDF) of gradient tensor sizes.

One solution is to fuse multiple small tensors together before performing all-reduce operation on them. After fusion, the size of the result tensor is the sum of sizes of small tensors, and the fused all-reduce operator is ready for execution when all the small all-reduce operators are ready. Though the fused tensor can potentially utilize the bandwidth better, its all-reduce start time is delayed, which affects the completion time of the DAG too. We design a dynamic programming approach to obtain a fused DAG balancing this trade-off in Sec. V.

III. System Overview

We consider distributed DNN training jobs using the all-reduce architecture in modern ML frameworks (*e.g.*, MXNet). We design a communication scheduler, *PACE*, to schedule all-reduce communication in DNN training for best overlapping communication with computation, in order to minimize model training time without affecting model quality.

Upon submission of a DNN training job to the ML cluster, *PACE* profiles the job and collects the following information (by running the training for a few iterations if there is no historical runtime trace): (i) the computation-communication DAG, (ii) execution time of each operator in the DAG, and (iii) size of the tensor to be all-reduced by each all-reduce operator (in bytes).

PACE then produces a good DAG for the job's training by fusing small all-reduce operators using the dynamic programming approach in Sec. V, and generates the optimal, preemptive execution schedule of all-reduce operators in the result graph using the optimization solution in Sec. IV.

After that, *PACE* uses the produced DAG and communication schedule to run the DNN training. All-reduce tensor

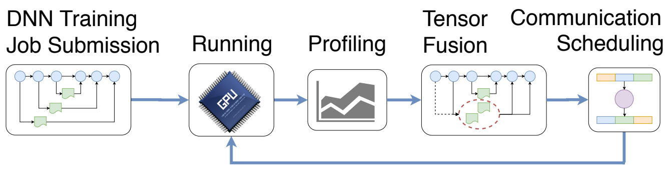


Fig. 5: *PACE* workflow.

fusion on the DAG and the communication schedule can be recomputed periodically if necessary, *e.g.*, when a significant change of execution time of operators is detected. Fig. 5 illustrates the workflow.

IV. Preemptive Communication Scheduling

Given the (fused) DAG, the communication scheduling module in *PACE* computes an optimal schedule of all all-reduce operators in the DAG, which minimizes the DAG's end-to-end execution time. The DAG is executed once per training iteration at each worker in a distributed DNN training job. Accelerating DAG execution speeds up job training.

A. Communication Scheduling Problem

DAG model. We consider a computation-communication DAG containing M computation operators and N all-reduce communication operators. A computation operator $j \in [M]$ ($[M] = \{1, \ldots, M\}$) may depend on zero or more other computation operators and zero or one all-reduce operator [8], and it can only be executed after all those dependent operators are done. Let P_j denote the set of computation operators that j depends on, and $P_j = \emptyset$ if no such dependency. v_j is the precedent all-reduce operator that j depends on, and $v_j = \emptyset$ if there is no such dependency. Suppose time is slotted into small time slots, *e.g.*, 5ms. Let f_j be the computation time of computation operator j (in terms of number of time slots). z_j is the completion time slot of computation operator j.

Each all-reduce operator $i \in [N]$ has a precedent computation operator u_i. u_i does not depend on any all-reduce operators directly or indirectly, *i.e.*, no all-reduce operator(s) among direct and indirect predecessors of u_i, which is true for all DNN training DAGs [8]. In some DNNs, one computation operator may generate multiple all-reduce tensors, such as weight and bias; we can treat them as one combined all-reduce tensor as their dependency and ready time are the same. The size of all-reduce tensor i is S_i.

Communication schedule. At each time slot t, we select at most one ready all-reduce operator to execute. Therefore, the length of a time slot should be set to be no larger than the execution time of the smallest (fused) all-reduce operator. Let $x_i(t)$ denote whether all-reduce operator i should be executed (*i.e.*, whether its tensor i should be all-reduced) at t: $x_i(t)$ is 1 if so, and 0, otherwise. Execution of an all-reduce operator may take more than one time slots, and we allow preemption of its execution. Therefore, the execution of an all-reduce operator may span multiple, none-necessarily consecutive time slots, *i.e.*, the schedule of all-reduce operator i can be described by $\{t | x_i(t) = 1, t \in [T]\}$. Let y_i denote

TABLE I: Notation

$x_i(t)$	whether to execute all-reduce operator i at time slot t		
z_j	completion time slot of computation operator j		
y_i	completion time slot of all-reduce operator i		
f_i	computation time of computation operator i		
S_i	tensor size of all-reduce operator i		
u_i	precedent computation operator of all-reduce operator i		
v_j	precedent all-reduce operator of computation operator j		
N	# of all-reduce communication operators		
M	# of computation operators		
P_j	set of precedent computation operators of computation operator j		
g_i	time to complete all-reduce on tensor i		
W	# of workers	T	# of time slots

the completion time slot of all-reduce operator i, which is the latest time slot with $x_i(t) = 1$.

As illustrated in Sec. II-C, non-consecutive execution of an all-reduce operator can be achieved by dividing its tensor into (variable size) partitions according to the computed schedule, and performing all-reduce on each tensor partition.

Execution time of all-reduce operator. Assume there are W workers in the training job and ring-based all-reduce is used [17]. The time to perform all-reduce on a tensor of size S_i is mainly spent on gradient transmission and gradient aggregation at the workers: each worker receives one gradient chunk while sending another at the same time; then the worker sums the received chunk with its own before sending/receiving the next chunk. Each worker transmits gradient chunks of size S_i/W for $2(W-1)$ times and aggregates these gradient chunks for $W-1$ times in total. Suppose the workers are homogeneous and synchronized. The time to complete ring all-reduce on tensor i is:

$$g_i = \frac{\frac{S_i}{W} \cdot 2(W-1)}{B} + \frac{\frac{S_i}{W}(W-1)}{G} \tag{1}$$

where B represents the bandwidth between workers and G is the computation power (*e.g.*, sum of floating points when using GPU). In practice, we collect sample points (tensor size, all-reduce time) during the profiling stage of a DNN job, and fit a linear model to describe the above relation between S_i and g_i. As shown in Fig. 3, such a linear model is quite accurate when the tensor size is sufficiently large. We focus on such cases in this section and will use the algorithm in Sec. V to obtain the DAG with small tensors fused.

Similarly, we can derive the execution time of all-reduce operators using other all-reduce algorithms, *e.g.*, binary tree, recursive doubling, as a function of tensor size [1].

Optimization problem. We formulate the following optimization problem to compute the best preemptive communication schedule in a given DNN training DAG. The main decision variables are $x_i(t), \forall i \in [N], t \in [T]$; y_i and z_j, $\forall i \in [N], j \in [M]$, are auxiliary variables; all the other quantities are input. The objective is to minimize the completion time of the latest computation operator(s) (all-reduce operators are typically not at the last of the DAG [8]), to minimize the DAG's execution time. Important notation is summarized in Table I for ease of reference.

$$\min \max_{j \in [M]} z_j \tag{2}$$

subject to:

$$\sum_{t \in [T]} x_i(t) \geq \lceil g_i \rceil, \forall i \in [N] \tag{3}$$

$$y_i = \max_{t \in [T]} \{t \mid x_i(t) = 1\}, \forall i \in [N] \tag{4}$$

$$z_j = f_j + \max_{k \in P_j} \{z_k, y_{v_j}\}, \forall j \in [M] \tag{5}$$

$$\sum_{i \in [N]} x_i(t) \leq 1, \forall t \in [T] \tag{6}$$

$$x_i(t) = 0, \forall i \in [N], t < z_{u_i} \tag{7}$$

$$x_i(t) \in \{0, 1\}, \forall i \in [N], t \in [T] \tag{8}$$

$$y_{v_j} = 0, \forall v_j = \emptyset \tag{9}$$

Constraint (3) ensures that for each all-reduce operator i, a sufficient number of time slots are allocated to accomplish its all-reduce. The right-hand side of (3) is computed according to the tensor size and all-reduce algorithm in use, *e.g.*, Eqn. (1) for ring all-reduce. Constraint (4) formulates the completion time slot of all-reduce operator i. As in (5), the completion time slot of computation operator j depends on the time when all its dependencies (precedent computation operators $k \in P_j$ and all-reduce operator v_j if it exists) are done and its own computation time. Constraint (6) ensures that at most one all-reduce operator is scheduled at each time slot. Constraint (7) ensures that an all-reduce operator i can only be scheduled after its precedent computation operator u_i is done.

We assume ideal computation operator scheduling in our communication scheduling problem above: there are sufficient computation devices such that each computation operator can immediately start execution once all its dependent operators are done (constraint (5)). Computation scheduling with device constraints is an open NP-hard problem [22], which is not addressed due to our focus on communication scheduling.

B. Optimal Solution

The optimization problem in (2) involves integer variables and non-conventional constraints in (4)(5). However, we are able to transform it into an equivalent form with a nice structure, which can be optimally solved in polynomial time.

Reducing constraints. Consider the example DAG on the left of Fig. 6. According to (5), we can compute recursively that (computation of other z_i's omitted due to space limit):

$z_7 = f_7 + \max\{z_6, y_6\} = y_6 + f_7$
(because $y_6 > z_6$ as all-reduce operator 6 depends on computation operator 6),

$$\begin{aligned} z_{12} &= f_{12} + \max\{z_9, z_{11}, y_1\} \\ &= \max\{y_1 + f_{12}, y_2 + f_9 + f_{12}, y_3 + f_8 + f_9 + f_{12}, \\ &\quad y_4 + f_{11} + f_{12}, y_5 + f_{10} + f_{11} + f_{12}, \\ &\quad y_6 + f_7 + \max\{f_8 + f_9, f_{10} + f_{11}\} + f_{12}\} \end{aligned}$$

The completion time of the last computation operator 12, z_{12}, is the longest time among the following: completion time of each all-reduce operator i, y_i, plus the overall computation time along the longest path from the computation operator j (which depends on i such that $v_j = i$) to the last computation operator, $\forall i \in \{1, \ldots, 6\}$. That is, $z_{12} = \max_{i \in \{1,\ldots,6\}} \{y_i + \sum_{j \in L_i} f_j\}$, where L_i is the longest computation path from the computation operator j with $v_j = i$ to the last computation operator. Since $y_i = \max_{t \in [T]} t x_i(t)$ according to (4), we then have $z_{12} = \max_{t \in [T], i \in \{1,\ldots,6\}} \{t x_i(t) + \sum_{j \in L_i} f_j\}$. We can prove that this is in general true for DNN training DAGs, as shown in Theorem 1 which uses Lemma 1.

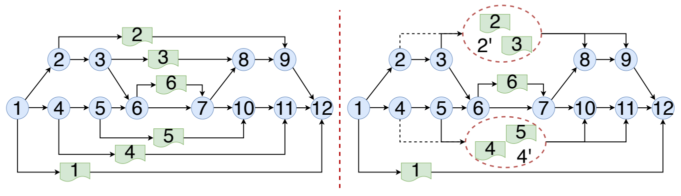


Fig. 6: An example DNN training DAG before and after communication tensor fusion.

Lemma 1. *As there is no all-reduce operator among direct and indirect predecessors of u_i ($\forall i \in [N]$) in a DNN training DAG, z_{u_i} can be directly computed based on (5) with computation time of its predecessor computation operators.*

Theorem 1. *The objective function in (2) with constraints (4)(5)(9) is equivalent to the following objective function:*

$$\max_{t \in [T], i \in [N]} \{t x_i(t) + \sum_{j \in L_i} f_j\}$$

where L_i is the longest computation path from the computation operator j with $v_j = i$ to the last computation operator.[1]

We prove Theorem 1 by induction on the last computation operator. In the basic case where there is only one computation operator, its finish time can trivially be presented as the form in Theorem 1. For the general case, we formulate the finish time of the last operator as a function of the latest finish time among its precedent operators. Then we merge the terms including the same y_j and express them in the form of the longest computation path, and derive the formula in Theorem 1 by induction hypothesis. All missing detailed proof of the paper is in our technical report.

Then the original problem (2) is equivalent to the following, only with decision variables $x_i(t), \forall i \in [N], t \in [T]$:

$$\min \max_{t \in [T], i \in [N]} \{t x_i(t) + \sum_{j \in L_i} f_j\} \tag{10}$$
$$s.t. \ (3)(6)(7)(8)$$

Converting to an ILP. We next transform the optimization problem (10) into an equivalent integer linear program (ILP). We will show that the ILP has a nice structure such that it is efficiently solvable.

We first transform the objective function in (10) into a convex function, exploiting a lexicographical order of the items $t x_i(t) + \sum_{j \in L_i} f_j, \forall t \in [T], i \in [N]$. We sort these NT items in non-increasing order and put them into a vector:

[1]If there are more than one computation operators with no successor in the DAG, we can add one dummy computation operator to be the last.

$$\alpha(\mathbf{x}) = (\ldots, tx_i(t) + \sum_{j\in L_i} f_j, \ldots)$$

$\alpha(\mathbf{x}')$ is lexicographically smaller than $\alpha(\mathbf{x})$, if $\alpha(\mathbf{x}')[k] < \alpha(\mathbf{x})[k]$ for the first index k where $\alpha(\mathbf{x}')[k]$ and $\alpha(\mathbf{x})[k]$ differ [23]. Then the objective function in (10) is equivalent to the following lexicographical minimization: $\text{lexmin}_{\mathbf{x}}\alpha(\mathbf{x})$, *i.e.*, finding the set of decision variables $\mathbf{x}^*$ such that $\alpha(\mathbf{x}^*)$ is lexicographically no larger than $\alpha(\mathbf{x})$ of any other $\mathbf{x}$. The rationale is that since $\alpha(\mathbf{x})$ is in non-increasing order, the value of the first item in $\alpha(\mathbf{x})$ is $\max_{t\in[T],i\in[N]}\{tx_i(t)+\sum_{j\in L_i} f_j\}$; $\mathbf{x}^*$ achieves the smallest $\max_{t\in[T],i\in[N]}\{tx_i(t)+\sum_{j\in L_i} f_j\}$, which is the optimal solution of (10).

We introduce a function $\phi(\alpha)$ to convert vector $\alpha(\mathbf{x})$ into a scalar, which preserves the lexicographical order (where $|\alpha| = TN$) [24]:

$$\phi(\alpha) = \sum_{k=1}^{|\alpha|} |\alpha|^{\alpha[k]} = \sum_{t\in[T],i\in[N]} (TN)^{tx_i(t)+\sum_{j\in L_i} f_j}$$

Then problem (10) is equivalent to the following problem:

$$\min_{\mathbf{x}}\ \phi(\alpha(\mathbf{x})) \qquad (11)$$
$$s.t.\ (3)(6)(7)(8)$$

where the objective function (11) is a summation of $(TN)^{tx_i(t)+\sum_{j\in L_i} f_j}$, which is a convex function of variable $x_i(t)$ since $TN > 0, t > 0$.

Next, we adopt the λ-representation technique [25] to transform (11) into an equivalent ILP. We have a set of $\lambda_i^0(t)$ and $\lambda_i^1(t) \in \mathbb{R}^+$, which satisfy:

$$x_i(t) = \lambda_i^1(t), \forall i \in [N], t \in [T] \qquad (12)$$
$$\lambda_i^0(t) + \lambda_i^1(t) = 1, \forall i \in [N], t \in [T] \qquad (13)$$

Then we can use linear function $(TN)^{\sum_{j\in L_i} f_j}\lambda_i^0(t) + (TN)^{t+\sum_{j\in L_i} f_j}\lambda_i^1(t)$ to approximate the integer convex function $(TN)^{tx_i(t)+\sum_{j\in L_i} f_j}$, and their values are proven to be equal with integer $x_i(t)$'s in [25]. Then we can convert (11) to the following ILP, and rest assured that their integer optimal solutions are the same:

$$\min_{\mathbf{x},\lambda} \sum_{t\in[T],i\in[N]} ((TN)^{\sum_{j\in L_i} f_j}\lambda_i^0(t) + (TN)^{t+\sum_{j\in L_i} f_j}\lambda_i^1(t)) \qquad (14)$$
$$\begin{aligned} s.t.\quad & (3)(6)(7)(8) \\ & (12)(13) \\ & \lambda_i^0(t), \lambda_i^1(t) \in \mathbb{R}^+, \forall i \in [N], t \in [T] \end{aligned}$$

Solving the ILP. The ILP in (14) is efficiently solvable due to the following:

Theorem 2. *The coefficient matrix of constraints (3)(6)(7)(12)(13) is a totally unimodular matrix.*

We prove that the coefficient matrix satisfies the criteria of total unimodularity [26]. It is known that for an ILP where constraint matrix is totally unimodular and the constants in the constraints are integers, there exists an optimal integer solution [26]. Therefore, we can solve the optimization problem in (14) without integer constraint (8) using any efficient LP solvers, such as interior-point methods, and obtain an optimal integer solution in polynomial time. The optimal solution to problem (14) is also an optimal solution to problem (2), *i.e.*, an optimal schedule of all all-reduce operators in the DAG.

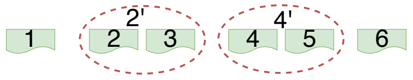


Fig. 7: Fusion of all-reduce operators in Fig. 6.

V. Balanced Communication Tensor Fusion

We now design an all-reduce tensor fusion algorithm for producing the DAG, which works together with optimal communication scheduling in Sec. IV to minimize iteration time.

A. Design Rationale

Our design strives to achieve a good trade-off between (a) efficient bandwidth utilization and (b) delay of execution start of fused all-reduce operators and their successor computation operators. For the example in Fig. 6, suppose all-reduce tensors 2 and 3 are small, and we merge them as a new all-reduce operator $2'$ for better bandwidth utilization. The new all-reduce operator $2'$ can not start execution until both computation operators 2 and 3 are done. Computation operators 8 and 9 cannot start before the fused tensor $2'$ is all-reduced.

To minimize the impact of (b), we fuse all-reduce operators with similar ready time, which leads to less start delay as compared to fusing operators that are far apart in ready time. To group tensors with similar ready time for fusion, we sort all-reduce operators in non-decreasing order of their ready time, *i.e.*, z_{u_i} (the completion time of predecessor computation operator u_i of all-reduce operator i), and construct a chain $\mathcal{N}$. Fig. 7 gives an example chain of all-reduce operators from Fig. 6, assuming $f_2 + f_3 < f_4$. We group adjacent all-reduce operators on the chain, and show the following:

Lemma 2. *Fusing adjacent all-reduce tensors, in non-decreasing order of their ready time, is better than fusing non-adjacent tensors, in terms of ready time of the fused tensor.*

The reason is obvious: given two groups for tensor fusion (as shown in Fig. 7), if we exchange two operators across the two groups, the ready time of the first fused tensor is delayed.

To achieve (a), we should ensure sizes of fused tensors are similar and sufficiently large, to lower-bound the worst performance of communication. This is related to two decisions: (i) the number of groups R, that we partition the operator chain into; (ii) the tensor-size-balanced group partition.

Given R, we design a dynamic programming approach to find balanced tensor groups, by maximizing the minimum total tensor size among partitioned groups:

$$\begin{aligned} & MaxMinSize(\mathcal{N}, R) \\ & = \max_{\mathcal{N}'\subset\mathcal{N}} \min\{\sum_{i\in\mathcal{N}'} S_i, MaxMinSize(\mathcal{N}\setminus\mathcal{N}', R-1)\} \end{aligned}$$

Here $\mathcal{N}'$ is a prefix subchain of $\mathcal{N}$, always starting from the first operator in the chain. We enumerate $\mathcal{N}'$ from containing the first all-reduce operator only to containing the first $N - R + 1$ operators. Given $\mathcal{N}'$, we let the rest of the chain to be partitioned into $R - 1$ balanced groups. We compare the total tensor size in $\mathcal{N}'$ ($\sum_{i \in \mathcal{N}'} S_i$) with the minimal size of fused tensors of the $R - 1$ maximally balanced groups, obtain the minimal fused tensor size among all R groups and identify the group partitions maximizing this minimal size. Finding the best $R - 1$ partitions within the subchain $\mathcal{N} \setminus \mathcal{N}'$ is a similar problem as partitioning $\mathcal{N}$ into R groups, except for at a smaller scale.

The number of tensor groups for fusion, R, cannot be too small, as otherwise the tensor groups are large and the start delay of fused tensors is significant. On the other hand, R cannot be too large, as then the fused tensors are still too small to fully utilize the bandwidth. We can find the best R by enumerating all possible values from 1 to N when N is small (*e.g.*, dozens), or using a stochastic search method such as Momentum-SGD [27] when N is large. Table II shows the number of all-reduce operators in 4 representative DNNs.

B. Tensor Fusion Algorithm

Alg. 1 shows how to find the best R (using the enumeration approach), produce R balanced tensor groups and generate the training DAG with potentially fused all-reduce tensors. At each R, we use the dynamic programming function **MaxMinSize** to produce the balanced R partitions along the ordered chain of all-reduce operators (line 4). Then we fuse the all-reduce operators in the original DNN training DAG accordingly (line 5), and compute the optimal communication schedule on the fused DAG using the optimization method in Sec. IV-B as an oracle (line 6). We compare the end-to-end execution time of the fused DAG achieved at different values of R, and identify the best R (and thus the fused DAG) that has the smallest end-to-end execution time (lines 7-9).

Algorithm 1 Balanced All-Reduce Tensor Fusion

```
Input: DNN Training DAG $G$, number of all-reduce operators $N$
    and computation operators $M$, computation time $f_j, \forall j \in [M]$,
    tensor size $S_i$, all-reduce time $g_i$, $\forall i \in [N]$
Output: Fused DAG and optimal communication schedule
 1: Order all all-reduce operators in $G$ in non-decreasing order of
    their ready time into chain $\mathcal{N}$.
 2: $min\_iter\_time = \infty, R^* = 0, \widehat{G}^* = \emptyset, h = \emptyset$
 3: for $R \in \{1, \dots, N\}$ do
 4:     $(maxmin\_t\_size, t\_groups) = \mathbf{MaxMinSize}(\mathcal{N}, R)$
 5:     Fuse all-reduce tensors according to obtained groups in
        $t\_groups$, and produce fused DAG $\widehat{G}$ (where the precedent
        computation operator of a fused all-reduce operator is specified
        as the one with the latest completion time among precedent
        computation operators of original tensors).
 6:     Compute the optimal communication schedule $h$ and end-to-
        end execution time $iter\_time$ of $\widehat{G}$ by solving (14).
 7:     if $iter\_time < min\_iter\_time$ then
 8:         $min\_iter\_time = iter\_time, R^* = R$
 9:         $\widehat{G}^* = \widehat{G}, h^* = h$
10: return fused DAG $\widehat{G}^*$ and optimal communication schedule $h^*$
11: function $\mathbf{MaxMinSize}(\mathcal{N}, R)$
12:     if $R = 1$ then
13:         return $\sum_{i \in \mathcal{N}} S_i$, $\{\mathcal{N}\}$
14:     else if $R >$ number of tensors in $\mathcal{N}$ then
15:         return $-\infty$, $\emptyset$
16:     $maxmin\_size = 0$, $best\_groups = \emptyset$
17:     for $\mathcal{N}' \subset \mathcal{N}$ do
18:         $t\_size, t\_groups = \mathbf{MaxMinSize}(\mathcal{N} \setminus \mathcal{N}', R - 1)$
19:         $temp\_size = \min(\sum_{i \in \mathcal{N}'} S_i, t\_size)$
20:         if $maxmin\_size < temp\_size$ then
21:             $maxmin\_size = temp\_size$
22:             $best\_groups = t\_groups \cup \{\mathcal{N}'\}$
23:     return $maxmin\_size$, $best\_groups$
```

Theorem 3. *Alg. 1 runs in polynomial time and produces optimally balanced all-reduce tensor fusion results.*

In the recursive function call (line 18), we store the results of $MaxMinSize(\mathcal{N}, R)$ computed at different $\mathcal{N}$ and R, to avoid re-computing the same subproblem in later iterations. Then line 4 can be done in polynomial time. We can also obtain the optimal scheduling solution from line 6 in polynomial time. So the entire Alg. 1 runs in polynomial time. The maximally balanced chain partition is ensured by the max-min dynamic programming function.

VI. Performance Evaluation

A. Implementation

We implement *PACE* using C++11 and Python>= 2.7 on top of Horovod [15], an open-source all-reduce communication library that uses NCCL [28] for executing ring all-reduce operations. We use MXNet 1.5 as the ML framework with Horovod for training DNNs. *PACE* wraps the *DistributedOptimizer* class in Horovod and intercepts all calls of ring all-reduce operations for executing our communication schedule. It receives tensor fusion instructions and optimal preemptive schedule of the (fused) tensors computed by Alg. 1, and implements communication scheduling as follows: it assigns a priority t to the (partitioned) all-reduce operator (due to preemptive scheduling), whose execution is to start from time slot t (a (partitioned) operator can be scheduled over multiple consecutive time slots), and a smaller value indicates higher priority; it enqueues these all-reduce operators into a priority queue and executes them according to their priorities. We also add a few lines of code to MXNet to collect the profiling trace (*e.g.*, the duration of each operator, the sizes of communication tensors).

B. Methodology

Testbed. We build a Kubernetes cluster [29] of 8 GPU servers connected by a Dell Z9100-ON switch. The peak bandwidth between any two servers is about 25Gbps, measured using iPerf [30]. Each server has one 8-core Intel E5-1660 CPU, two GTX 1080Ti GPUs, 48GB RAM, and one MCX413A-GCAT 50GbE NIC. We installed NVIDIA GTX driver 384.90, CUDA 9.0, CuDNN 7.0 and NCCL 2.4.7 in each Ubuntu server.

Benchmark models. We experiment with jobs training 2 CNN models, VGG16 [19] and ResNet50 [20], on ImageNet dataset [31], and 2 RNN models, Seq2Seq [32] and Transformer [21], on WMT dataset [33] (see Table II for details). The training scripts are from the official MXNet

TABLE II: Benchmark models

DNN Models	# of parameters (Million)	# of all-reduce operators	# of computation operators
VGG16	138.4	32	80
ResNet50	25.5	157	335
Seq2Seq	67.2	32	293
Transformer	91.8	164	1061

examples [34], [35]. Each worker process has 1 GPU and batch sizes per GPU for the 4 models are 64, 32, 4096 and 2048 samples, respectively. We use training speed (images/sec for CNNs and tokens/sec for RNNs) as the performance metric, which is computed by $\frac{1}{\text{training time per iteration}} \times$ batch size, where the training time per iteration is the end-to-end execution time of the model's DAG. All the reported speed numbers are averaged over 1000 training iterations.

Baselines. We compare *PACE* with three representative baselines: (i) WFBP [3], default communication optimization implemented in most ML frameworks (*e.g.*, MXNet, PyTorch), which overlaps all-reduce communication with gradient calculation and schedules ready all-reduce operators using FIFO, without tensor fusion. (ii) Default Horovod [15], the state-of-the-art all-reduce communication framework which integrates a few optimization techniques, such as WFBP and fusion of small tensors. (iii) TicTac [5], a heuristic communication scheduler for PS architecture in TensorFlow, which greedily reduces the blocking time of activation calculation by changing the communication order. P3 [12] adopts a similar idea.

C. Worker Number vs. Speed

We train each DNN using different numbers of workers in a 10Gbps network (we limit the available bandwidth using Linux tc tool), which is a typical bandwidth setting for AWS instances (*e.g.*, g3.4xlarge GPU instance) [36]. In Fig. 8, we observe desirable linear scaling of the total training speed achieved by *PACE* with the worker number. We see that *PACE* outperforms WFBP by 20%-36%, default Horovod by 3%-32% and TicTac by 13%-21% across the 4 benchmark models.

PACE achieves much higher speed than Horovod on the models except ResNet50. This is because ResNet50 has fewer parameters (see Table II) and the all-reduce time in a 10Gbps network is much shorter than the computation time, leaving little room for improvement with communication scheduling. On the contrary, other models are more communication-intensive and benefit significantly from *PACE*.

The outperformance of *PACE* over TicTac is mainly due to its better communication granularity, as TicTac only considers communication order without tensor fusion nor partition. For large models such as VGG16 and Transformer, *PACE* achieves a higher speedup due to preemptive scheduling, which avoids the head-of-line blocking issue caused by large tensors (*e.g.*, two largest tensors in VGG16 are 67MB and 410MB). For ResNet50 with a lot of small tensors, *PACE* fuses them to mitigate all-reduce overhead and better utilize the network.

D. Bandwidth v.s. Speed

We investigate how much performance gain *PACE* can achieve at different bandwidth levels. In this experiment, we use 12 workers to train each model.

In Fig. 9, we observe that on our testbed, the training speed becomes stable when the network bandwidth is larger than 10-15Gbps, as then the bandwidth is no longer the bottleneck in further reducing the communication time, while the overhead of ring all-reduce (*e.g.*, negotiation time) is. *PACE* tends to achieve its highest speed at a smaller bandwidth level than the others, revealing its more efficient bandwidth usage and overlap of communication with computation.

PACE outperforms Horovod by up to 32% for VGG16 at 12Gbps network bandwidth and 27% for Transformer at 14Gbps. Training of models with fewer parameters also benefits from *PACE* when communication time is more important: as compared to Horovod, *PACE* achieves 32% speed-up with ResNet50 at 6Gbps bandwidth.

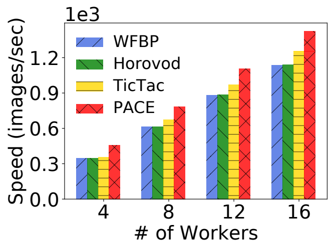
When the bandwidth is higher, the speed-up by *PACE*, as compared to the baselines, may become smaller. *PACE* outperforms Horovod by 13% and 9% when training VGG16 and Transformer at 20Gbps bandwidth, respectively; they achieve similar speeds with ResNet50 and Seq2Seq. Communication takes less time when bandwidth is higher, which is especially so for small models such as ResNet50 and Seq2Seq, leaving *PACE* less room for improvement. On the other hand, when bandwidth is very small, all-reduce communication time dominates training iteration time, and overlapping communication with computation brings little gain. This explains the similar training speed achieved by all strategies at very low bandwidth.

We note that the training speed and performance gain that *PACE* can achieve are decided by both network bandwidth and computation capacity. On our testbed, we use one GTX 1080Ti GPU per worker and the highest gain can be achieved when network bandwidth is about 10-15Gbps. In a (commercial) AI cloud equipped with more powerful computation devices (*e.g.*, Tesla V100 GPU which is 2x-9x faster than GTX 1080Ti), the highest gain can be reached at much higher bandwidth levels (*e.g.*, 50-100Gbps) due to the much faster computation.

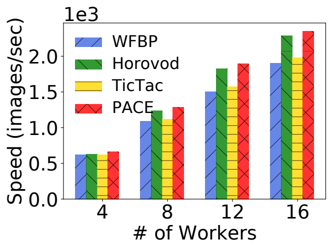
E. Bandwidth Usage

We further examine the bandwidth usage when training each model using 12 workers. We measure the outbound traffic in worker 0 at the NIC level using Linux ethtool every 5 milliseconds. Inbound traffic and outbound traffic are the same across workers in ring all-reduce.

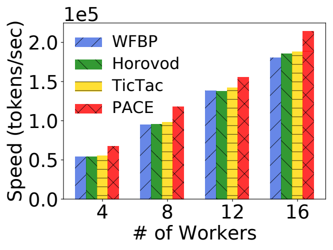
In Fig. 10, we observe that both *PACE* and Horovod experience regular peaks (*i.e.*, communication burst) and valleys (*i.e.*, network idle) in network usage. The duration of the cycle is the time of one training iteration, with the valley happening during activation computation and the peak indicating large tensor transmission. Compared to Horovod, *PACE* significantly improves bandwidth usage by nearly fully overlapping communication with computation in most models, as shown by the very short duration of valleys of only a few milliseconds. In the special case of Seq2Seq model, the heaviest communication operations are near the input of the DNN, and all all-reduce operations have already been completed when the latter part of



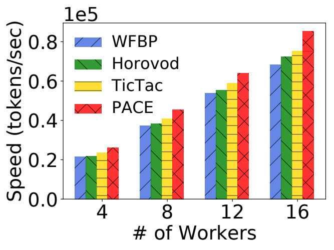
(a) VGG16



(b) ResNet50

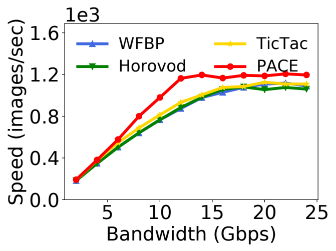


(c) Seq2Seq

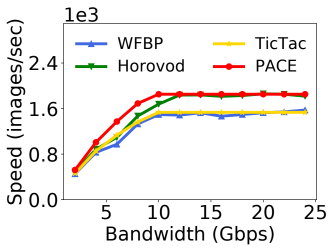


(d) Transformer

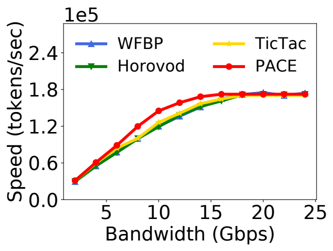
Fig. 8: Training speed with different numbers of workers per job.



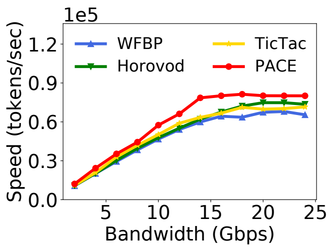
(a) VGG16



(b) ResNet50

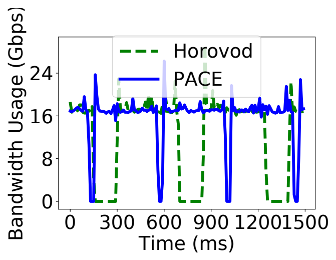


(c) Seq2Seq

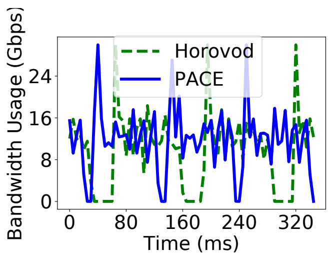


(d) Transformer

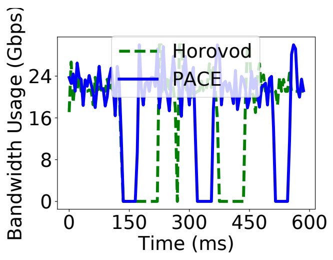
Fig. 9: Training speed at different bandwidth levels.



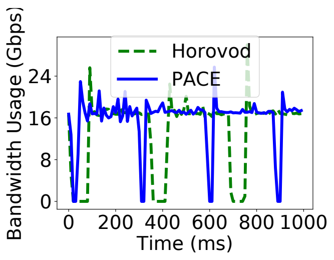
(a) VGG16



(b) ResNet50



(c) Seq2Seq



(d) Transformer

Fig. 10: Network bandwidth usage during training.

the model begins activation calculation. This leaves *PACE* less room for overlapping communication with more computation.

VII. Related work

Communication acceleration. Existing communication optimization approaches include: (1) speeding up individual messages using high performance network, *e.g.*, RDMA [7], or GPU-optimized collective library, *e.g.*, NCCL [28]; (2) reducing data traffic by leveraging techniques such as gradient quantization [4] [37] [38] and sparse parameter synchronization [39]; (3) mitigating stragglers by adopting load-aware and interference-aware resource scheduling [40]–[43]; (4) reducing synchronization overhead by allowing stale parameter updates, *e.g.*, Bounded Staleness Parallel [44] or Round-Robin Synchronous Parallel [45]; (5) improving network-layer performance by using multicast or flow control [46]. These efforts are complementary to our work.

Overlapping communication with computation. Most DNN frameworks, *e.g.*, TensorFlow, PyTorch, MXNet, Poseidon and B-Caffe, support overlapping communication with gradient calculation. P3 [12] further overlaps parameter synchronization with forward propagation using priority scheduling in MXNet PS architecture. TicTac [5] proposes a similar idea but without preemption and tensor fusion, and shows a smaller training speed-up (less than 20%) in TensorFlow PS architecture. iBatch [13] batches parameter pulls in BigDL PS architecture. Instead, *PACE* focuses on the all-reduce architecture. We adopt the natural DAG representation in modern ML frameworks and address issues caused by sub-optimal granularity of communication tensors using preemptive communication scheduling and tensor fusion.

VIII. Conclusion

PACE is an all-reduce communication scheduler for accelerating DNN training. At its core is a theoretically optimal algorithm of preemptive communication scheduling for DAG representation in modern ML frameworks. We also design a dynamic programming approach to fuse small tensors to improve bandwidth utilization. Through extensive evaluation on multiple DNNs with different training settings in our GPU testbed, we demonstrate that *PACE* can achieve near-optimal overlapping of communication with computation, with up to 36% improvement in terms of training speed when compared to state-of-the-art communication scheduling approaches.

References

[1] S. Shi, X. Chu, and B. Li, "MG-WFBP: Efficient Data Communication for Distributed Synchronous SGD Algorithms," in *Proc. of IEEE INFOCOM*, 2019.

[2] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le *et al.*, "Large Scale Distributed Deep Networks," in *Proc. of NIPS*, 2012.

[3] 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. of USENIX ATC*, 2017.

[4] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding," in *Proc. of NIPS*, 2017.

[5] S. H. Hashemi, S. A. Jyothi, and R. H. Campbell, "TicTac: Accelerating Distributed Deep Learning with Communication Scheduling," in *Proc. of SysML*, 2019.

[6] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh *et al.*, "Mixed Precision Training," in *Proc. of ICLR*, 2018.

[7] J. Liu, J. Wu, and D. K. Panda, "High Performance RDMA-based MPI Implementation over InfiniBand," *International Journal of Parallel Programming*, 2004.

[8] 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. of USENIX OSDI*, 2016.

[9] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic Differentiation in PyTorch," in *Proc. of NIPS Autodiff Workshop*, 2017.

[10] 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," in *Proc. of NIPS Workshop on Machine Learning Systems*, 2016.

[11] 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 ACM SOSP*, 2019.

[12] A. Jayarajan, J. Wei, G. Gibson, A. Fedorova, and G. Pekhimenko, "Priority-Based Parameter Propagation for Distributed DNN Training," in *Proc. of SysML*, 2019.

[13] S. Wang, A. Pi, and X. Zhou, "Scalable Distributed DL Training: Batching Communication and Computation," in *Proc. of AAAI*, 2019.

[14] "PaddlePaddle," http://en.paddlepaddle.org/en, 2019.

[15] A. Sergeev and M. Del Balso, "Horovod: Fast and Easy Distributed Deep Learning in TensorFlow," *arXiv preprint arXiv:1802.05799*, 2018.

[16] L. Bottou, "Large-scale Machine Learning with Stochastic Gradient Descent," in *Proc. of COMPSTAT*, 2010.

[17] P. Patarasuk and X. Yuan, "Bandwidth Optimal All-reduce Algorithms for Clusters of Workstations," *Journal of Parallel and Distributed Computing*, 2009.

[18] P. Hintjens, *ZeroMQ: Messaging for Many Applications*, 2013.

[19] A. Z. Karen Simonyan, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in *Proc. of ICLR*, 2015.

[20] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proc. of IEEE CVPR*, 2016.

[21] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is All You Need," in *Proc. of NIPS*, 2017.

[22] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, "Device Placement Optimization with Reinforcement Learning," in *Proc. of ICML*, 2017.

[23] P. C. Fishburn, "Exceptional Paper—Lexicographic Orders, Utilities and Decision Rules: A Survey," *Management Science*, 1974.

[24] Z. Huang, B. Balasubramanian, M. Wang, T. Lan, M. Chiang, and D. H. Tsang, "Need For Speed: Cora Scheduler for Optimizing Completion-Times in the Cloud," in *Proc. of IEEE INFOCOM*, 2015.

[25] R. Meyer, "A Class of Nonlinear Integer Programs Solvable by a Single Linear Program," *SIAM Journal on Control and Optimization*, 1977.

[26] K. Bernhard and J. Vygen, *Combinatorial Optimization: Theory and Algorithms*, 6th ed. Springer, 2018.

[27] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the Importance of Initialization and Momentum in Deep Learning," in *Proc. of ICML*, 2013.

[28] "NVIDIA Collective Communications Library (NCCL)," https://developer.nvidia.com/nccl, 2019.

[29] "Kubernetes," https://kubernetes.io/, 2019.

[30] iPerf, "Run Deep Learning with PaddlePaddle on Kubernetes," https://iperf.fr/, 2019.

[31] "ImageNet Dataset," http://www.image-net.org, 2017.

[32] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to Sequence Learning with Neural Networks," in *Proc. of NIPS*, 2014.

[33] "WMT 2017," http://www.statmt.org/wmt17/, 2017.

[34] "MXNet Image Classification," https://github.com/apache/incubator-mxnet/tree/master/example/image-classification, 2019.

[35] "MXNet sockeye," https://github.com/awslabs/sockeye, 2019.

[36] "AWS EC2 Instance," https://aws.amazon.com/ec2/instance-types/, 2019.

[37] 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 *Proc. of NIPS*, 2017.

[38] M. K. Hyeontaek Lim, David G Andersen, "3LC: Lightweight and Effective Traffic Compression for Distributed Machine Learning," in *Proc. of SysML*, 2019.

[39] A. F. Aji and K. Heafield, "Sparse Communication for Distributed Gradient Descent," in *Proc. of EMNLP*, 2017.

[40] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters," in *Proc. of ACM EuroSys*, 2018.

[41] Y. Bao, Y. Peng, C. Wu, and Z. Li, "Online Job Scheduling in Distributed Machine Learning Clusters," in *Proc. of IEEE INFOCOM*, 2018.

[42] Y. Peng, Y. Bao, Y. Chen, C. Wu, C. Meng, and W. Lin, "DL2: A Deep Learning-driven Scheduler for Deep Learning Clusters," *arXiv preprint arXiv:1909.06040*, 2019.

[43] Y. Bao, Y. Peng, and C. Wu, "Deep Learning-based Job Placement in Distributed Machine Learning Clusters," in *Proc. of IEEE INFOCOM*, 2019.

[44] 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 *Proc. of NIPS*, 2013.

[45] C. Chen, W. Wang, and B. Li, "Round-robin Synchronization: Mitigating Communication Bottlenecks in Parameter Servers," in *IEEE INFOCOM*, 2019.

[46] L. Mai, C. Hong, and P. Costa, "Optimizing Network Performance in Distributed Machine Learning," in *Proc. of USENIX HotCloud*, 2015.