Title Addressing Network Bottlenecks with Divide-and-Shuffle Synchronization for Distributed DNN Training

Authors Wang, Weyian; Zhang, Cengguang; Yang, Liu; Chen, Kai; Tan, Kun

Source Proceedings - IEEE INFOCOM, v. 2022-May, May 2022, article number 9796688, p. 320-329

Conference 41st IEEE Conference on Computer Communications, INFOCOM 2022, Virtual; London, UK, 2 - 5 May 2022, Code 180260

Version Accepted Version

DOI 10.1109/INFOCOM48880.2022.9796688

Publisher IEEE

Addressing Network Bottlenecks with Divide-and-Shuffle Synchronization for Distributed DNN Training

Weiyan Wang[1], Cengguang Zhang[1], Liu Yang[1], Kai Chen[1], Kun Tan[2]
[1]iSING Lab, Hong Kong University of Science and Technology, [2]Huawei

***Abstract*—Bulk synchronous parallel (BSP) is the de-facto paradigm for distributed DNN training in today's production clusters. However, due to the global synchronization nature, its performance can be significantly influenced by network bottlenecks caused by either static topology heterogeneity or dynamic bandwidth contentions. Existing solutions, either system-level optimizations strengthening BSP (e.g., Ring or Hierarchical Allreduce) or algorithmic optimizations replacing BSP (e.g., ASP or SSP, which relax the global barriers), do not completely solve the problem, as they may still suffer from communication inefficiency or risk convergence inaccuracy.**

In this paper, we present a novel divide-and-shuffle synchronization (DS-Sync) to realize communication efficiency without sacrificing convergence accuracy for distributed DNN training. At its heart, by taking into account the network bottlenecks, DS-Sync improves communication efficiency by dividing workers into non-overlap groups to synchronize independently in a bottleneck-free manner. Meanwhile, it maintains convergence accuracy by iteratively shuffling workers among different groups to ensure a global consensus. We theoretically prove that DS-Sync converges properly in non-convex and smooth conditions like DNN. We further implement DS-Sync and integrate it with PyTorch, and our testbed experiments show that DS-Sync can achieve up to 94% improvements on the end-to-end training time with existing solutions while maintaining the same accuracy.

***Index Terms*—distributed DNN training, synchronization, communication efficiency, convergence accuracy**

I. Introduction

In today's production training clusters, BSP is the *de-facto* paradigm to train DNN models with large datasets across different workers [1]–[4]. In every iteration, BSP enforces a global synchronization to aggregate gradients from all workers and then distribute them back to each worker so that it updates model parameters as exactly the same way of training on the single machine [5]. However, there are network bottlenecks caused by either physical network oversubscription or dynamic bandwidth contentions in real-world environments. The synchronization process for some workers may be significantly affected (as shown in Fig. 1 (a)), incurring considerable idle waiting as a result of the global barrier.

Thus, in the production network environment, taking BSP as the benchmark, an ideal synchronization scheme for distributed DNN training should achieve the following goals:

- **Communication efficiency:** it should be topology-aware and fully utilize network bandwidth while avoiding bottlenecks to reduce idle waiting of BSP in each iteration.
- **Convergence accuracy:** it should maintain the same convergence accuracy as BSP in similar iterations, which is widely considered as the best in this aspect [6]–[9].

Existing solutions, no matter whether system-level optimizations strengthening BSP (e.g., Ring [10] or Hierarchical Allreduce [11]) or algorithmic optimizations replacing BSP (e.g., asynchronous parallelism (ASP) [12] or stale synchronous parallelism (SSP) [13]), do not achieve the above two goals simultaneously (details in §II-B). For the former one, while solutions like Ring [10], Tree [14], or Hierarchical Allreduces [11] explore decentralized collective methods to improve bandwidth utilization, they still have a long dependency chain that may block downstream communications if network bottlenecks exist. Furthermore, due to their global synchronization nature, these solutions can hardly avoid idle waiting. For the latter one, by relaxing the global synchronization barrier with ASP [12] or SSP [13], they avoid or defer idle waiting of BSP in each iteration. However, these methods often incur a slower convergence speed, i.e., requiring a larger number of training iterations. Particularly, as network bottlenecks accumulate the staleness (iteration gap between the fastest and the slowest worker) on the same worker, it can bring in outdated and noisy gradients, leading to convergence inaccuracy.

To this end, we propose DS-Sync, a new divide-and-shuffle synchronization to achieve both communication efficiency and convergence accuracy simultaneously (§III). The key idea of DS-sync is to divide workers into non-overlap groups of different sizes to synchronize independently and periodically shuffle workers among groups to reach global consensuses. According to the network topology and situations, DS-Sync generates a periodical pattern to divide and shuffle all workers. In every iteration, any worker optimizes its model with local gradients and then synchronizes and averages the model parameters with workers in the same group. In the next iteration, workers are shuffled among groups in the generated pattern so that any worker synchronizes parameters with some unseen workers from different groups in the previous iteration. Fig. 1 compares our DS-Sync with PS (widely used in BSP, ASP, and SSP) and the topology-aware Hierarchical Allreduce to illustrate the advantages of DS-Sync.

DS-Sync improves communication efficiency with *dividing* and ensures convergence accuracy with *shuffling*. By taking network bottlenecks into account, it divides workers into different groups to minimize the maximum communication cost of all groups to reduce idle waiting. The insight is that the more workers to synchronize, the more communication traffic and the more latency (§II-B1). DS-Sync always keeps bottlenecked workers or links in the smaller group to speed up, while other regular ones form larger groups. Therefore, bottlenecked workers can catch up with others to reduce idle waiting

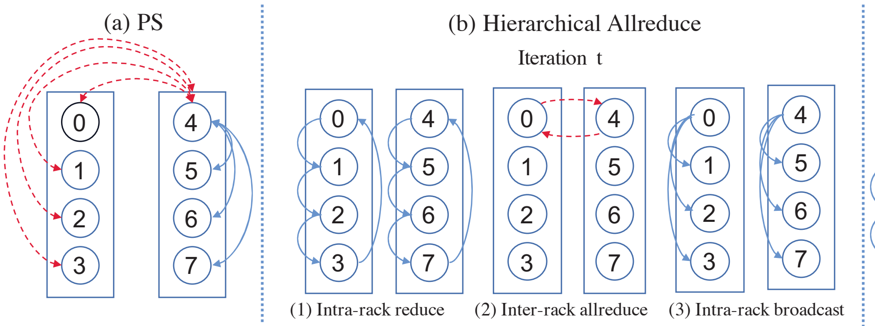

Fig. 1: Comparison in the setting of two racks with upper-level links indicated by red dashed lines. (a) In PS(widely used in BSP, ASP, and SSP), some workers are in different racks from the PS in worker 4, thus suffering from inter-rack bottlenecks when pushing to or pulling from the PS. (b) Topology-aware Hierarchical Allreduce decomposes the global communication into three serial steps, including intra-rack reduce, inter-rack allreduce, and intra-rack broadcast. While it mitigates traffic over the inter-rack bottleneck, it is still a global synchronization process of three serial communication steps. (c) DS-Sync divides workers into different sized groups to synchronize independently and shuffles group members every iteration. It fully utilizes intra-rack bandwidth while reducing the traffic and communication latency over the bottleneck. More importantly, all the groups synchronize *in parallel* as opposed to *sequentially* as the Hierarchical Allreduce.

and improve communication efficiency. DS-Sync maintains convergence via iterative shuffling, which is formally proven in §IV-B. DS-Sync carefully designs the shuffling pattern to ensure that any worker can directly or indirectly exchange information with all the groups during the shuffling period. Intuitively, local updates of all groups in any iteration are iteratively merged in the upcoming iterations. As a result, all workers can reach global consensus iteratively instead of global synchronization in every iteration. Furthermore, DS-Sync extends the stochastic weight average (SWA) to ensemble the diverse local models for better generalization performance.

We have implemented DS-Sync in the PyTorch framework and conducted comprehensive testbed experiments (§V). Our results show that DS-Sync can improve communication efficiency without any loss in convergence speed or accuracy. For example, compared to the Allreduce BSP, DS-Sync can achieve up to 94% improvement in terms of end-to-end training time to reach the target accuracy. In the scenario of bandwidth contention, DS-Sync improves communication efficiency by up to 2X while maintaining the same accuracy as BSP with a similar number of iterations.

Overall, this paper makes three key contributions:

1) We propose DS-Sync, a new divide-and-shuffle synchronization scheme, to achieve both communication efficiency and convergence accuracy simultaneously.
2) We prove that DS-Sync converges to the same accuracy of BSP with a similar number of iterations under the nonconvex and smooth conditions of DNN.
3) We implement DS-Sync in PyTorch and validate its effectiveness through extensive testbed experiments.

II. Backgrounds and Motivations

In this section, we first overview the network bottlenecks in production clusters and then discuss the drawbacks of existing solutions in handling them.

A. *Network Bottlenecks in Training Cluster*

Fig. 2 illustrates a typical architecture of a training cluster shared by different tasks, including computation nodes, storage nodes, and the network. The computation nodes have accelerators like GPUs or TPUs but with limited storage as the local cache. The storage nodes with RAID work as the elastic and fast Network File System (NFS). The network usually adopts hierarchical physical topology like spine-leaf, which is scaled up easily by adding switches in each layer [15]. All nodes are grouped into racks, each connected by a leaf switch. All leaf switches are connected to upper-level spine switches. In this setting, network bottlenecks may arise due to:

1) **Static topology heterogeneity:** Intra-rack communication is non-blocking, but inter-rack communication depends on the inter-rack link load and oversubscription ratio [2]. If a task has multiple inter-rack connections, it can suffer from the oversubscription problem.
2) **Dynamic inter-rack bandwidth contention:** In production clusters, there are often background flows from other training tasks that compete for the inter-rack bandwidth [1]–[3] (e.g. the leaf 1 in Fig. 2). Regular pairs of workers can deliver over 2x the throughput of those delayed by the inter-rack bandwidth contention [3].
3) **Dynamic end-host bandwidth contention:** Different distributed training tasks can co-locate in the same physical node to occupy different GPUs or TPUs but share the same NIC [1], [2] (e.g. the 3rd node of leaf 1 in Fig. 2). This causes end-host bandwidth contention that potentially slows down the communication of corresponding workers.

B. *Existing Solutions and Their Drawbacks*

BSP can be significantly influenced by the above network bottlenecks. In general, there are two categories of existing works to improve it. One is the system-level optimizations

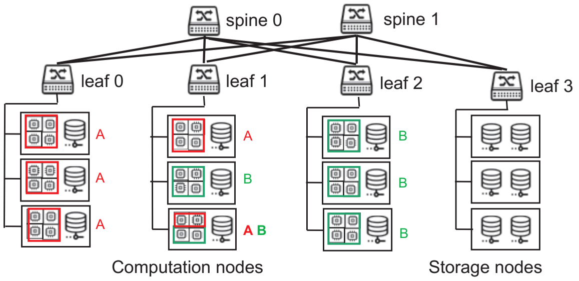

Fig. 2: An example of GPU cluster shared by tasks A and B.

strengthening BSP [3], [5]–[11], [14], [16]–[18], the other is algorithmic optimizations replacing BSP [12], [13], [19]–[21]. However, neither of them can achieve the aforementioned two goals simultaneously in production clusters.

1) System-level Optimization: Due to its global synchronization nature, BSP can hardly eliminate the idle waiting of regular workers and links. Some system-level works employ various topologies like PS [5], [16], Ring [10], and double tree [14] to fully use network bandwidth. Some others explore different underlying network optimizations, including overlapping communication and computation [6]–[9], [17], [18], RDMA [22], [23], in-network aggregation [24]–[26], congestion control [27], [28], flow scheduling [29]–[31], and coflow scheduling [32]–[34]. However, all these system-level works are either topology-agnostic and/or contention-vulnerable.

Both heterogeneous topologies and dynamic contentions lead to network bottlenecks. Unaware of the network topology[1], these works cannot be guaranteed to align the logical topology with the physical one. Multiple connections may cross and compete for the inter-rack links, resulting in the network bottleneck of oversubscription. Additionally, Ring Allreduce and Tree Allreduce introduce long dependency in the logical topology and pipelines. They are vulnerable to bandwidth contentions on inter-rack links or end-host NIC since they can easily block downstream communications. Hence, the network bottleneck brings in significantly idle waiting time for BSP.

Topology-aware Hierarchical Allreduces are still suboptimal since the network bottleneck essentially stalls regular workers and links. Hierarchical Allredcue [3], [11], [35] decomposes the global synchronization into sequential communications to localize the network bottleneck. Specifically, Blueconnect [11] makes three serial communications steps, including intra-rack reduce, inter-rack allreduce, and intra-rack broadcast. The serial communications make all workers related with intra-rack communication wait for the inter-rack communication and vice versa. Plink [3] takes a further step to slice data into chunks and uses a pipeline to overlap inter-rack and intra-rack communications. However, some workers simultaneously participate in inter-rack and intra-rack communications, which is the bottleneck slowing down communications in other pipeline stages.

TABLE I: Communication cost comparison

	Latency	Transfer Delay
PS	$2(N-1)L\alpha$	$2(N-1)S\beta/P$
Ring AR	$2(N-1)L\alpha$	$2(N-1)S\beta/N$
Double Tree AR	$2(\log N+k)L\alpha$	$2(\log N+k)S\beta/k$

P stands for the PS number, and k is the data chunk number.

Even in an ideal and uniform network, the BSP communication time increases with the total number of workers. Specifically, the communication cost of point-to-point transferring a model of S bytes can be modeled as $L\alpha + S\beta$ [36], where α denotes the latency, β is the transfer delay for one byte, and L is the number of times to invoke model synchronization layer by layer [6]–[9] to overlap computation and communication. The more workers N to synchronize, the more point-to-point communications and total communication traffic, as summarized in Tab. I for popular BSP methods. We also conduct experiments to verify it in different worker numbers and different models, and Fig. 3 shows the trend that communication time increases with worker numbers, which motivates us to divide all workers and put the bottleneck in the small enough one to speed up.

2) Algorithmic Optimization: Previous algorithmic works only focus on occasional stragglers due to stochastic events like system interruption. This approach offers more flexibility by relaxing the synchronization conditions, expecting the straggler to happen on another worker later. However, workers delayed by the network bottleneck accumulate the staleness to a high level, which is beyond their expectations and results in convergence and throughput problems.

ASP [12] does not have the guarantee to converge at all. It directly gets rid of all synchronizations and hopes that all workers have similar paces. However, the accumulated high-level staleness due to network bottlenecks is against it. Although ASP has no waiting time, the consistent outdated gradients due to network bottlenecks on the same worker usually result in lower convergence accuracy.

Due to accumulated staleness, SSP [13], [19] suffers from convergence inaccuracy and communication inefficiency. It defers global synchronization until exceeding staleness bound. If the bound is high, the consistent staleness leads to a slower

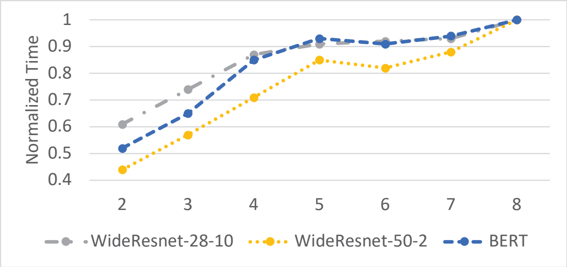

Fig. 3: Normalized communication time vs. worker number (normalized by the maximum time value)

[1] Current Allreduce implementations like NCCL and MPI are unaware of physical network topology. NCCL only detects different physical link types within the node such as NVlink, PCI-E, and network.

Pseudo Code 1: The workflow of DS-Sync

```
1  import DS_Sync
2  def train(dataloader, i=rank):
3    #detect the physical topology and initialize the
       group list for DS-Sync accordingly
4    topology = DS_Sync.topologyDetect()
5    groupList = DS_Sync.initGroups(topology)
6    #initialize all wokrers' models in the same way
7    model, lossCriteron, optimizer, SWA_model =
       DS_Sync.model_generator()
8    for t in range(Total_iteration):
9      #adjust the grouping according to dynamic
       bandwidth variations in every 100 iterations
10     if t%period==0 And bandwidth changes:
11       groupList = DS_Sync.adjustGroups(groupList)
12     #FP and BP computation on the sampled data batch
13     x, y = DataLoader()
14     pred = model(x)
15     loss = LossCriteron(pred, y)
16     gradient = loss.backward()
17     #locally update model parameters
18     optimizer.step()
19     optimizer.zero_grad()
20     #pick the proper group to synchronize for worker
        i in the iteration t
21     group = groupList[t%len(groupList)][i]
22     #synchronize and average the parameters within
       the same group and record bandwidth
23     model = DS_Sync.Average(model, group)
24     #track and record communication cost for the
       group adjustment
25     DS-Sync.commRecord()
26     #local stochastic weight average ensemble
27     SWA_model = factor*SWA_model+ (1-factor)*model
28   #weight average all local SWA_models to ensemble
       all trajectories as the final model to test
29   Final_model = AverageAllreduce(model, world)
```

convergence rate and even failures sometimes [37]. Otherwise, the staleness also frequently invokes global synchronization with low bound, resulting in idle waiting, just like BSP.

Originated from IoT network, Gossip [20], [21], [38] suffers from inefficient point-to-point communications and unawareness of network bottlenecks. It breaks the global barrier into chained local barriers. Every worker only sends and receives messages to the neighbors in the sparsely connected graph to relay information iteratively. Gossip has the congestion problem in the end host if there are over two peers in a worker's neighborhood [38]. The connected graph of Gossip can be time-variant, such as the dynamic exponential graph [21] and random matching [38]. Hence, there can be many connections to worsen the inter-rack contention in some iterations. Furthermore, the longer chain of gossip propagation, the slower it is to reach consensus [21].

III. Design

This section presents the detailed design of DS-Sync. First, we introduce the overall workflow of DS-Sync. Then, we discuss how workers are divided and shuffled to handle different network bottlenecks.

A. Overall Workflow

At its heart, by taking network bottlenecks into account, DS-Sync periodically divides and shuffles workers to form multiple non-overlapping groups of different sizes to synchronize independently. DS-Sync first senses the physical network topology and bandwidth contentions, and then it generates the periodical divide-and-shuffle pattern accordingly (details in §III-B). DS-Sync synchronizes and averages workers' model parameters in the same group in every iteration according to the divide-and-shuffle pattern. DS-Sync also extends SWA to ensemble diverse trajectories of different workers. DS-Sync can be easily integrated with the DL framework like PyTorch as stated in Pseudo Code 1. Specifically, DS-Sync has the following key steps:

1) **Topology Detection** `topologyDetect()`: While the network topology can be known to the operators, it is often unknown to the general users. Fortunately, network profiling such as DPDK and iPerf can be employed to measure the round-trip latency and bandwidth between worker pairs. The topology information can thus be derived from the measurements [3].
2) **DS-Sync Group Initialization** `initGroups()`: According to the static topology heterogeneity, DS-Sync initializes the periodical divide-and-shuffle pattern that divides workers into the inter-rack group and intra-rack groups as described in §III-B1. In this way, DS-Sync can minimize its own connections and communication traffic crossing the inter-rack links to avoid oversubscriptions.
3) **DS-Sync Group Adjustment** `adjustGroups()`: Once sensing bandwidth contention due to background flow changes in a period, DS-Sync further adjust the periodical divide-and-shuffle pattern. Since the bottlenecked group is caused by bandwidth contention on either inter-rack link or end-host NIC, the related inter-rack or intra-rack group can be further divided into smaller ones. Therefore, DS-Sync decreases the max group communication cost and keeps all groups in similar paces to proceed.(details in §III-B2 III-B3).
4) **Parameter Synchronization** `code line 12-24`: DS-Sync synchronizes all groups in parallel. Within the group, it uses Allreduce to average the parameters of workers in the group after they locally optimize their model parameters. The model parameter contains all information about the past parameter updating steps. In the next iteration, DS-Sync shuffles workers among groups so that different groups can exchange the past updating information with each other. Specifically, the model parameter of worker i in iteration T is $w_T^{(i)} = w_0 + \sum_{t=0}^{T} \prod_{k=t}^{T} W_k e_i \triangle_t$, where $\triangle_t$ is local update vector of all workers in the iteration t, W_k is the group average in the iteration K, e_i is a unit vector that only the i-th element is 1, and $\prod_{k=t}^{T} W_k e_i \triangle_t$ is the approximated distributed average. DS-Sync also tracks every workers' communication cost to adjust group patterns later.
5) **Extended SWA Ensemble** `code line 25-28`: DS-Sync can further exploit the diverse parameter trajectories of different workers to improve test accuracy by

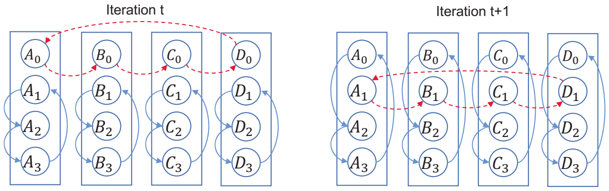

Fig. 4: Group initialization divides all workers into inter-rack and intra-rack groups and shuffles workers to be the representative in the inter-rack group by turns.

ensembling. Every worker applies exponential moving average on local parameters in every iteration to get a local SWA model [39]. At the end of the training, all workers average their SWA models to ensemble all trajectories to reach a more flatten optimal solution. The flatten optimal solution is more robust to small data perturbation than a sharp one, which generalizes better on unseen test data [40].

DS-Sync improves communication efficiency by dividing workers into groups of different sizes and guarantees convergence with shuffling workers among groups to reach consensus iteratively. According to the network situation, DS-Sync always keeps bottlenecked workers or links in smaller groups, while other regular ones form larger groups. Then delayed workers or links have less latency and communication traffic than others. In this way, bottlenecked workers in small groups can catch up and reduce the idle waiting of others. Additionally, DS-Sync shuffles workers among groups in every iteration. It follows the periodical divide-and-shuffle pattern guaranteeing that the past local updates of different workers can be merged by iterative propagation. Therefore, any worker can directly or indirectly get the past global updating information iteratively instead of immediately like BSP. Finally, all workers can reach global consensus iteratively and maintain the convergence on training data (formal proof in § IV-B). DS-Sync also extends SWA to ensemble all diverse local models for better generalization on test data.

B. Handling Different Network Bottlenecks

Now, we introduce how DS-Sync periodically divides and shuffle workers into groups according to different network bottlenecks. To achieve both goals, there are two principles for generating the divide-and-shuffle group pattern:

- It should decouple network bottlenecks from others and keep them in a smaller group to alleviate bottlenecks.
- It should guarantee that all workers can directly or indirectly exchanges information by shuffling.

DS-Sync generates divide-and-shuffle group patterns in the group initialization and adjustment. In the initialization, DS-Sync forms inter-rack and intra-rack groups according to the static network topology. Then during the training, it reacts to the dynamic bandwidth contention by further dividing the bottlenecked group into multiple smaller groups. Specifically, DS-Sync handles three kinds of network bottleneck as follows:

1) Static Topology Heterogeneity: After detecting the physical network topology in `topologyDetect()`, DS-Sync initializes the group pattern in `initGroups()` by dividing workers into the inter-rack groups and intra-rack groups. Firstly, all workers in the same rack form intra-rack groups. DS-Sync further separates one worker from each intra-rack group to be the representative forming an inter-rack group. In the shuffling, all workers in the same rack take turns to be the representative. Since Allreduce is used for synchronization within any group, DS-Sync can guarantee itself has only one connection crossing one inter-rack link. Therefore, it is free of the oversubscription problem caused by itself. Every intra-rack group exchanges members with the inter-rack group to reach the global consensuses iteratively. Fig. 4 illustrates an example of four racks how workers are divided and shuffled in inter-rack and intra-rack groups.

2) Inter-rack Link Bandwidth Contention: If any rack is sensed to have slow inter-rack communication due to background flows from other tasks in the same rack, DS-Sync adjusts the inter-rack group in `adjustGroups()`. It further divides the inter-rack group and keeps the influenced representative in the smallest groups. Specifically, DS-Sync put each delayed representative with another regular representative to form an inter-rack group of two workers, while other regular representatives form a large inter-rack group. In every iteration, the regular representative in the small inter-rack group is exchanged with another representative in the large inter-rack group. Workers in the same rack also take turns to be the representative to get global information exposed. Fig. 5 shows an example that the inter-rack group is further divided into two inter-rack groups.

3) End-host NIC Bandwidth Contention: If DS-Sync finds a worker has smaller bandwidth due to sharing the end-host NIC with another task in the same node, `adjustGroups()` also adjust the group pattern accordingly. DS-Sync always keeps the delayed worker in the intra-rack group. If there are enough workers in the rack hosting the delayed worker, DS-Sync further divides the intra-rack group into sub ones. Similar to §III-B2, DS-Sync pair the delayed worker with another regular worker to form a small intra-rack group, and the rest workers form a large intra-rack group. The large intra-rack group exchanges workers with the small group as well as the inter-rack group. Therefore, the large group works as an

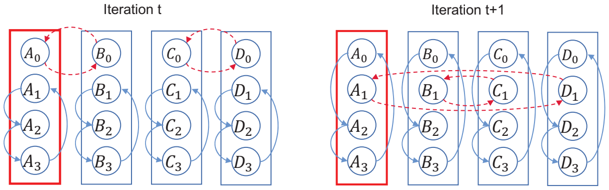

Fig. 5: Rack A (in the bold and red rectangles) has some background flows from other tasks to share inter-rack bandwidth. DS-Sync further divides inter-rack groups to reduce the group size in group adjustment during training.

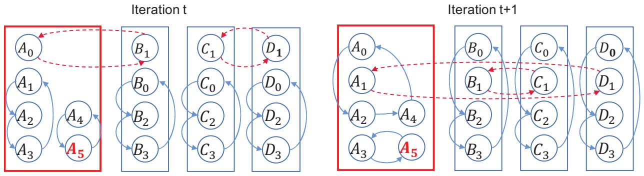

Fig. 6: In rack A, worker A_5 (in the bold and red characters) has another task sharing the end-host NIC. The upper-level link of rack A also has smaller bandwidth due to background flows. Besides the inter-rack group, DS-Sync further divides the intra-rack group to alleviate the bottleneck on worker A_5.

information hub of the rack so that all subgroups can exchange information.

IV. Analysis

In this section, we start with the quantitative communication analysis on DS-Sync and other methods in the scenario of the inter-rack bottleneck. Then, we mathematically prove that DS-Sync has the same convergence accuracy and rate as BSP in the nonconvex and smooth conditions of DNN.

A. Communication Time Analysis

For simplicity, we analyze the communication time in a simplified and flattened hierarchical network. A total number of N workers are evenly allocated into C racks, and each rack has G nodes. We assume that the same background flows of other tasks interfere with the available inter-rack bandwidth. α is the latency for one hop, β_1 denotes intra-rack transfer delay for one byte, and β_2 strands for inter-rack transfer delay for one byte. We also assume that all methods perfectly align their logical topologies with the physical network topology to minimize their inter-rack connections and communication traffic. The DNN model is synchronized for L times to overlap communication and computation.

Tab II summarizes the communication cost consisting of latency and transfer delay for various previous methods and our DS-Sync. PS is widely used in BSP, ASP, and SSP, but it suffers from congestion on the inter-rack link because of high crossing rack traffic. Due to the long dependency in logical topologies and pipelines, Ring Allreduce and Tree Allreduce are vulnerable to the contention of background flows. If an inter-rack link is bottlenecked, it slows down communication in the other stages. Hierarchical Allreduce has multiple serial communication steps so that the communication cost is the sum of all steps. If it further slices data into chunks to transfer in the pipeline, the workers in both inter-rack and intra-rack levels slow down the whole pipeline due to simultaneously sending and receiving messages in different stages. In Gossip algorithms, each rack has two or even more inter-rack connections in some iterations for the time-varied graph. Any worker also suffers from the in-cast problem if it has multiple neighbors. Due to the dependency of chained local barriers, bottlenecked workers finally stall others.

TABLE II: Communication Cost Summary

Methods	Latency	Transfer Delay
PS	$2(N-1)L\alpha$	$\frac{2[(C-1)GS\beta_2+(G-1)S\beta_1]}{P}$
Ring	$2(N-1)L\alpha$	$2(N-1)S\beta_2/N$
Double Tree	$2(logN+k)L\alpha$	$2(LogN+k)S\beta_2/k$
Hierarchical	$2(G+C-2)L\alpha$	$\frac{2(G-1)S\beta_1}{G}+\frac{2(C-1)S\beta_2}{C}$
Gossip	$2L\alpha$	$\geq 4S\beta_2$
DS-Sync	$2L\alpha$ or $2L(G-1)\alpha$	$2S\beta_2$ or $\frac{2(G-2)S\beta_1}{G-1}$

DS-Sync keeps the bottlenecked links and related workers in the smallest group of two workers, and $G-1$ is the size of the intra-rack group.

In contrast, our DS-Sync divides all workers into multiple non-overlap groups to synchronize in parallel. According to the network topology and situations, DS-Sync always assigns bottlenecked workers and links in smaller groups to catch up. The total communication time of DS-Sync is determined by the max one of all inter-rack and intra-rack groups.

B. Convergence Analysis

We mathematically prove that DS-Sync converges properly in nonconvex and smooth conditions of DNN. Firstly, we state some common assumptions of nonconvex and smooth DNN. Then we prove how DS-Sync can achieve global consensuses on model parameters iteratively. Finally, we show that DS-Sync can converge properly in nonconvex and smooth conditions.

1) Assumptions: Following the previous convergence theory for SGD in nonconvex and smooth conditions [41], we make some common assumptions in the optimization community for DNN as follows:

Assumption IV.1. Lipschitzian smooth: Any local function of worker i $F_i(\cdot)$ is with L-Lipschitzian gradients.

$$\|\triangledown F_i(x;\xi)-\triangledown F_i(y;\xi)\| \leq L\,\|x-y\|$$

Assumption IV.2. Bounded variance: Assume the variance of stochastic gradient $\mathbb{E}_{i\sim\mathcal{U}([n])}E_{\xi\sim\mathcal{D}_i}\left\|\triangledown F_i(x;\xi)-\triangledown f(x)\right\|^2$ is bounded for any parameters x with worker i uniformly sampled from $\{1,\cdots,n\}$ and data batch ξ from the distribution $\mathcal{D}_i$. This implies there exist constants σ such that

$$\mathbb{E}_{i\sim\mathcal{U}([n])}E_{\xi\sim\mathcal{D}_i}\left\|\triangledown F_i(x;\xi)-\triangledown f(x)\right\|^2 \leq \sigma^2$$

2) Consensus Proof: Before formally analyzing the convergence, we first prove that DS-Sync can reach the global consensus for the distributed average problem iteratively. In distributed average problem, each node i starts with a number $x_0^{(i)}$, and X_k stands for the vector $[x_k^{(0)}, x_k^{(1)}, ..., x_k^{(N)}]^T$ in the iteration k, and the W_k is a N-by-N transition matrix in $X_{k+1}=W_kX_k$, which indicates how workers exchange parameters in the iteration k. The goal of distributed average is to approximate the average $\bar{x}=\frac{1}{N}\sum_i x_0^{(i)}$ in every worker by K iterations as $X_K=W_{K-1}W_{K-2}\cdots W_0X_0$.

In DS-Sync, any W_k represents how workers are divided into multiple non-overlap groups to be synchronized and averaged in the iteration k. The entry $w_{i,j}$ is either 1/group size or 0, which means worker i and j are in the same group or not. The W_k is the symmetric doubly stochastic matrix, in which every entry is not negative, and the sum of any row or column is one. Unlike the transition matrix of the Gossip representing a sparsely connected graph, the W_k of DS-Sync stands for multiple fully connected sub-graphs covering all nodes without overlapping nodes and edges, which decouples network bottlenecks from others. Furthermore, W_k follows the periodical divide-and-shuffle pattern such that $W_k = W_{k+B}$ and $\prod_{i=0}^{B} W_i$ indicates a connected graph of all nodes. It can be easily calculated that the second largest eigenvalue $\rho := \lambda_2(\prod_{k=lB}^{(l+1)B-1} W_k) < 1$. According to previous works in distributed average [42], the worst-case rate of convergence can be related to the second-largest as stated in Lemma IV.1.

Lemma IV.1. We can bound the average error in worker i for DS-Sync as

$$\left\| \frac{\mathbf{1}_N}{N} X_0 - \prod_{k=0}^{kB} W_k e_i X_0 \right\|^2 \leq \rho^k \left\| X_0 \right\|^2 \qquad \forall i \text{ and } k \geq B$$

3) Convergence Proof: The formal proof of DS-Sync convergence is based on the iteratively reached consensus on historical updates and the similarity of recent updates from smooth assumptions of DNN. The model parameter of the worker can be rewritten as $w_T^{(i)} = w_0 + \sum_{t=0}^{T} \prod_{k=t}^{T} W_k e_i \triangle_t$, in which $\triangle_t$ is local update vector of all workers in the iteration t and $\prod_{k=t}^{T} W_k e_i \triangle_t$ is its approximated average. According to Lemma IV.1, the approximated average in worker i for the historical update $\bar{\triangle}_t$ is bounded as $\left\| \bar{\triangle}_t - \prod_{k=t}^{T} W_k e_i \triangle_t \right\|^2 \leq \rho^{(T-t)/B} \left\| \triangle_t \right\|^2$. DS-Sync can achieve better global consensus on earlier updates during the training. Although the recent update information may not be approximated very well, the smooth conditions of DNN bound the variances of gradients from different workers. Intuitively, different workers have slight differences in their local model parameters since most early past updating information has been averaged properly. According to the assumption of Lipschitzian smooth and bounded variance, the similar model parameters of different workers should have similar expectations of stochastic gradients on randomly sampled data batches. Therefore, we can formally prove the convergence of DS-Sync in the following theorems.

Theorem IV.2. (Convergence of DS-Sync). Based on Assumption IV.1 and IV.2 and Lemma IV.1, we show that the gradients of all workers converge to be 0 with the same rate $O(1/\sqrt{K})$ as BSP [41]. In other words, DS-Sync can reach a minimal point in the nonconvex and smooth case like BSP. Specifically, if the total number of iterate K is sufficiently large, then it is bounded as follows:

$$\frac{\sum_{k=0}^{K-1} \mathbb{E} \left\| \triangledown f(\frac{X_k \mathbf{1}_N}{N}) \right\|^2}{K} \leq \frac{8(f(0) - f^*)L}{K} + \frac{(8f(0) - 8f^* + 4L)\sigma}{\sqrt{Kn}}$$

V. Evaluation

In this section, we first introduce how DS-Sync and baselines are implemented as well as the experiment settings. Then, we discuss the experimental results of DS-Sync in various scenarios with network bottlenecks to seek answers to the following questions:

1) **Can DS-Sync achieve end-to-end performance advantages in various scenarios?** Our extensive evaluation verifies that DS-Sync has up to 94% improvements over the baselines in terms of the end-to-end training time in different network bottleneck scenarios.
2) **Can DS-Sync alleviate different network bottlenecks to improve communication efficiency?** In the three network bottleneck scenarios, DS-Sync consistently achieves minimal communication time, which improves the efficiency by up to 2x over BSP baselines.
3) **Can DS-Sync maintain a similar convergence rate and accuracy as BSP?** Different from ASP or SSP, DS-Sync can achieve the same accuracy as BSP with a similar number of iterations in the experiments. Especially for the smaller dataset, DS-Sync can have slightly higher accuracy due to the SWA ensemble.

A. Implementation and Experiment Settings

1) Implementation: We implement DS-Sync and integrate it (as well as BSP baselines, including Allreduce BSP, Hierarchical Allreduce BSP, ASP (can only use PS [5]), and Gossip (dynamic exponential graph)) with PyTorch. We choose the NCCL library as the communication backend for all methods and leave itself to determine the proper Allreduce topologies (Tree by default) for BSP baselines and group synchronization of DS-Sync. Since ASP cannot use collective operators, we implement it with send and receive primitives. We register the backward hook and pre-forward hook for every layer to overlap communication and computation. In the backward hook, we conduct a local optimizer update and invoke the non-block synchronization layer by layer. In the pre-forward hook of the corresponding layer, we set the barrier waiting for the synchronization event to make sure updating is completed before forwarding computation.

2) Evaluation Metrics: We evaluate the following metrics to verify that DS-Sync can simultaneously achieve the two goals mentioned earlier:

- **Time-to-Accuracy (TTA):** We use it to measure end-to-end training time. The target accuracy is set to the lowest one of BSP in five times running. We omit ASP results due to its failure to reach the target accuracy.
- **Communication Time:** We measure the communication efficiency as the time from invoking the first non-block communication to all communications completed (For ASP, we only report the fastest worker).
- **Iteration Number:** We use the iteration number of reaching the target accuracy to measure the convergence rate. We omit it for ASP again for the same reason.
- **Best Accuracy:** We record the best accuracy or F1 during the training to measure the convergence accuracy. Except for ImageNet, we run all experiments five times to get the mean and standard variation of best accuracy.

3) Models and Training Settings: We evaluate *WideResnet-28-10* [43] on **CIFAR10/100** [44], *WideResnet-50-2* [43] on **ImageNet** [45], and *BERT* [46] on **SQuADv1.1** [47] in the fields of Computer Vision and Natural Language Processing.

In principle, we follow the common practices [43], [46] in the machine learning (ML) community to split data, pre-process data, and set hyper-parameters. For DS-sync and all baselines, all hyper-parameters are the same as the original ML model papers except that we use the warm-up learning rate schedule for the large overall batch size. Due to the limited space, we do not list the detailed hyper-parameters here.

4) Testbed: We use a private cluster of 10 GPU nodes as the experiment environment. Each physical node has NVIDIA V100 GPUs, Intel Xeon Silver 4114 CPU, 191 GB memory, and a NIC. All nodes are divided into two racks and connected in the spine-leaf network. The network has 20Gb Ethernet per link, and its oversubscription rate is 1.

5) Network Bottleneck Settings: We manually align logical topologies of all baselines with the physical one in the best way to minimize inter-rack connections. We can run another background task training the BERT model distributedly on two workers of 2 racks to create background flows sharing the inter-rack link and end-host NIC. Specifically, we conduct experiments with the following different network bottlenecks:

- **Inter-rack and end-host bandwidth contentions:** The background task has one worker in the same node but not the same GPU with our target task, which shares both the inter-rack and end-host bandwidth. Our task has eight workers in different nodes of two racks, including worker 0-4 in rack 0 and worker 5-7 in rack 1. One worker of the background task co-locates in the same physical node with worker 4, while the other worker is in rack 1.
- **Inter-rack bandwidth contentions:** The background task has workers in the same rack but not the same node with our target task, which only shares the inter-rack bandwidth. Our task divides workers evenly into two racks: worker 0-3 in rack 0 and worker 4-7 in rack 1. The background task has two workers in the rest nodes of the two racks.
- **Static topology heterogeneity:** There is no other background task causing bandwidth contention but only static topology heterogeneity. Our task still divides workers evenly into two racks, including worker 0-3 in rack 0 and worker 4-7 in rack 1.

B. Experiment Results

The experiment results verify that DS-Sync has the best communication efficiency without any loss in convergence rate and accuracy in various scenarios of different network bottlenecks.

1) Inter-rack and End-host Bandwidth Contention: In this scenario, DS-Sync first divides workers into inter-rack and intra-rack groups in the initialization. During the training, it further divides the intra-rack group in rack 1 into two smaller ones. DS-Sync keeps the influenced worker 4 in the group of 2 workers. By measuring the TTA, DS-Sync achieves the best end-to-end performance, as shown in Fig. 7.

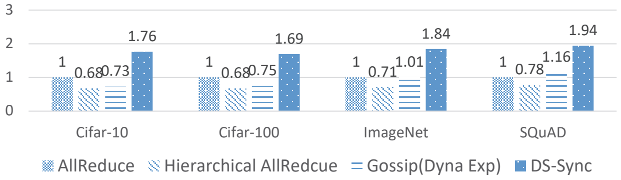

Fig. 7: Speed up on TTA in inter-rack and end-host bandwidth contention

TABLE III: Summary on communication efficiency and convergence in inter-rack and end-host bandwidth contention

Dataset	Methods	Comm. Time(ms)	Iter. #	Best Acc.
Cifar10	AR	540.1	6560	92.51±0.92%
	Hier. AR	790.2	6560	92.51±0.92%
	ASP(PS)	>567.3	-	87.46±1.31%
	Gossip	772.8	6360	92.47±1.17%
	DS-Sync	**338.1**	5640	92.62±0.56%
Cifar100	AR	545.6	12740	76.79±0.28%
	Hier. AR	801.5	12740	76.79±0.28%
	ASP(PS)	>573.8	-	66.23±2.03%
	Gossip	778.3	12130	76.66±0.32%
	DS-Sync	**341.8**	11460	76.76±0.35%
ImageNet	AR	825.7	341,664	76.27%
	Hier. AR	1078.2	341,664	76.31%
	ASP(PS)	>690.2	-	72.72%
	Gossip	834.5	341,664	76.10%
	DS-Sync	**412.5**	341,664	76.32%
SQuAD	Methods	Comm. Cost(ms)	Iter. #	Best F1
	AR	3260.7	7,400	92.66±0.08%
	Hier. AR	4213.2	7,400	92.66±0.08%
	ASP(PS)	>2741.7	-	89.85±1.21%
	Gossip	2805.5	7,400	92.61±0.1%
	DS-Sync	**1636.6**	7,400	92.64±0.05%

Furthermore, we summarize all detailed metrics on communication efficiency and convergence for all methods in Tab. III. Results show that DS-Sync gets the best communication efficiency by keeping inter-rack and end-host bottlenecks in the smallest independent groups. Meanwhile, it does not sacrifice the convergence rate and accuracy compared with BSP methods in all datasets.

However, other baselines suffer from communication inefficiency or convergence inaccuracy. The smaller one of inter-rack and end-host bandwidth is the bottleneck for the whole Allreduce pipeline. Since Hierarchical Allreduce does not handle the contention in end-host NIC, its serial communications are even slower than Allreduce. The PS architecture of ASP and the dynamic exponential graph of Gossip can have multiple inter-rack connections, which worsens the inter-rack network bottlenecks. Furthermore, ASP fails to reach the target accuracy because accumulated high staleness brings in consistent outdated gradients harmful to the convergence.

2) Inter-Rack Bandwidth Contention: DS-Sync keeps the bottlenecked inter-rack link in the group of 2 workers to alleviate the inter-rack network bottleneck. Fig. 8 illustrates that DS-Sync also achieves the best end-to-end performance to reach the target accuracy in the scenario of inter-rack bandwidth contention.

We evaluate all metrics on communication efficiency and convergence for all methods in this scenario, as shown in Tab. IV. The results verify that DS-Sync has the best commu-

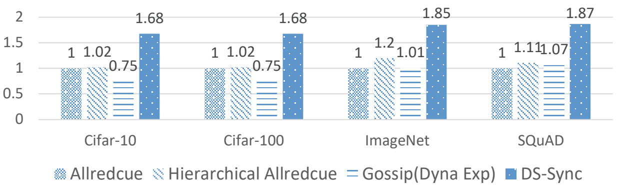

Fig. 8: Speed up on TTA in inter-rack bandwidth contention

TABLE IV: Summary on communication efficiency and convergence with inter-rack bandwidth contention

Dataset	Methods	Comm. Time(ms)	Iter. #	Best Acc.
Cifar10	AR	525.1	6560	92.51±0.92%
	Hier. AR	515.3	6560	92.51±0.92%
	ASP(PS)	>561.2	-	88.17±1.58%
	Gossip	764.1	6360	92.47±1.17%
	DS-Sync	**331.7**	5640	92.79±0.56%
Cifar100	AR	531.6	12740	76.79±0.28%
	Hier. AR	523.5	12740	76.79±0.28%
	ASP(PS)	>581.5	-	68.19±2.51%
	Gossip	768.3	12130	76.66±0.32%
	DS-Sync	**336.1**	11460	76.86±0.31%
ImageNet	AR	823.1	341,664	76.27%
	Hier. AR	685.2	341,664	76.31%
	ASP(PS)	>701.9	-	73.50%
	Gossip	812.9	341,664	76.10%
	DS-Sync	**407.4**	341,664	76.38%
SQuAD	Methods	Comm. Time(ms)	Iter. #	Best F1
	AR	3129.3	7,400	92.66±0.08%
	Hier. AR	2801.5	7,400	92.66±0.08%
	ASP(PS)	>2783.1	-	90.27%±0.51%
	Gossip	2913.1	7,400	92.61±0.1%
	DS-Sync	**1633.4**	7,400	92.68±0.05%

nication efficiency by reducing inter-rack communication traffic and simultaneously synchronizing all inter-rack and intra-rack groups. Meanwhile, it also reaches the same accuracy as BSP methods within a similar iteration number.

In contrast, other baselines do not handle the bottleneck so well. The inter-rack bandwidth contention slows down the whole Allreduce pipeline. Although Hierarchical Allreduce has better communication efficiency by isolating the inter-rack bottleneck, it cannot simultaneously conduct inter-rack and intra-rack communication like our DS-Sync. ASP and Gossip can still worsen the inter-rack bottleneck. Once again, ASP fails to converge properly due to the high staleness.

3) Static Topology Heterogeneity: In the group initialization, DS-Sync divides workers into one inter-rack group and two intra-rack groups to synchronize in parallel and avoid the oversubscription problem. Fig. 9 shows that DS-Sync still reaches the target accuracy in the minimum time in the static hierarchical topology without any contention.

Tab. V summarizes all metrics on communication efficiency and convergence for all methods for the static hierarchical topology. It shows that DS-Sync has advantages in communication efficiency since it synchronizes multiple smaller groups in parallel instead of the global one. DS-Sync also maintains the same convergence accuracy and rate as BSP methods.

The baselines cannot simultaneously realize both aforementioned goals as DS-Sync does. Allreduce takes a longer time for global synchronization. Since Allreduce is free of oversubscriptions, Hierarchical Allreduce pays a higher communication cost than Allredcue for serial communications. ASP in PS architecture and Gossip in dynamic exponential graphs suffer from the oversubscription problem due to their multiple inter-rack connections. Additionally, the convergence of ASP is still affected by the oversubscription.

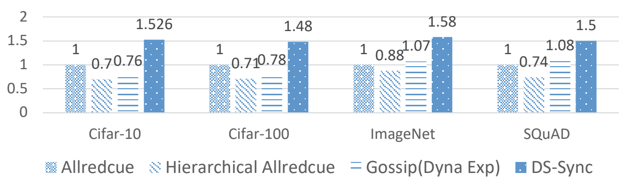

Fig. 9: Speed up on TTA in static topology heterogeneity

TABLE V: Summary on communication efficiency and convergence in static topology heterogeneity

Dataset	Methods	Comm. Time(ms)	Iter. #	Best Acc.
Cifar10	AR	266.1	6560	92.51±0.92%
	Hier. AR	390.8	6560	92.51±0.92%
	ASP(PS)	>637.3	-	88.85±1.45%
	Gossip	376.9	6360	92.47±1.17%
	DS-Sync	**200.1**	5640	92.79±0.56%
Cifar100	AR	271.3	12740	76.79±0.28%
	Hier. AR	395.1	12740	76.79±0.28%
	ASP(PS)	>641.2	-	69.41±1.81%
	Gossip	374.8	12130	76.66±0.32%
	DS-Sync	**198.1**	11460	76.86±0.31%
ImageNet	AR	441	341,664	76.27%
	Hier. AR	506.6	341,664	76.31%
	ASP(PS)	>785.5	-	73.70%
	Gossip	405.3	341,664	76.10%
	DS-Sync	**249.2**	341,664	76.38%
SQuAD	Methods	Comm. Time(ms)	Iter. #	Best F1
	AR	1631.8	7,400	92.66±0.08%
	Hier. AR	2233.4	7,400	92.66±0.08%
	ASP(PS)	>3146.9	-	90.30%±0.47%
	Gossip	1502.6	7,400	92.61±0.1%
	DS-Sync	**1060.2**	7,400	92.68±0.05%

VI. Conclusions

We proposed the new DS-Sync to address network bottlenecks in the production cluster. DS-Sync divides and shuffles workers to form groups of different sizes. The groups are non-overlapping and synchronized in parallel. In theory, we quantitatively analyze its communication cost and mathematically prove its convergence. Furthermore, we conduct extensive testbed experiments to validate that DS-Sync can achieve communication efficiency and convergence accuracy simultaneously.

Acknowledgment

This work is supported in part by the Hong Kong RGC TRS T41-603/20-R, GRF-16215119, GRF-16213621 and a Huawei research grant. We thank Li Chen and Hong Zhang for the helpful discussion and the anonymous reviewers for their constructive feedback and suggestions. Kai Chen is the corresponding author of the paper.

REFERENCES

[1] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *ATC*, 2019.

[2] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis C. M. Lau, Yuqi Wang, Yifan Xiong, and Bin Wang. Hived: Sharing a GPU cluster for deep learning with guarantees. In *OSDI*, 2020.

[3] Liang Luo, Peter West, Arvind Krishnamurthy, Luis Ceze, and Jacob Nelson. Plink: Discovering and exploiting locality for accelerated distributed training on the public cloud. In *MLSys*, 2020.

[4] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *OSDI*, 2018.

[5] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *OSDI*, 2014.

[6] Ammar Ahmad Awan, Khaled Hamidouche, Jahanzeb Maqbool Hashmi, and Dhabaleswar K. Panda. S-caffe: Co-designing MPI runtimes and caffe for scalable deep learning on modern GPU clusters. In *SIGPLAN*, 2017.

[7] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *ATC*, 2017.

[8] Yixin Bao, Yanghua Peng, Yangrui Chen, and Chuan Wu. Preemptive all-reduce scheduling for expediting distributed DNN training. In *INFOCOM*, 2020.

[9] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed DNN training acceleration. In *SOSP*, 2019.

[10] Amith R. Mamidala, Jiuxing Liu, and Dhabaleswar K. Panda. Efficient barrier and allreduce on infiniband clusters using multicast and adaptive algorithms. In *CLUSTER*, 2004.

[11] Minsik Cho, Ulrich Finkler, David S. Kung, and Hillery C. Hunter. Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. In *MLSys*. mlsys.org, 2019.

[12] Benjamin Recht, Christopher Ré, Stephen J. Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NeurIPS*, 2011.

[13] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. More effective distributed ML via a stale synchronous parallel parameter server. In *NeurIPS*, 2013.

[14] Pitch Patarasuk and Xin Yuan. Bandwidth efficient all-reduce operation on tree topologies. In *IPDPS*, 2007.

[15] Kashif Bilal, Samee Ullah Khan, Joanna Kolodziej, Limin Zhang, Khizar Hayat, Sajjad Ahmad Madani, Nasro Min-Allah, Lizhe Wang, and Dan Chen. A comparative study of data center network architectures. In *ECMS*, 2012.

[16] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *OSDI*, 2020.

[17] Jiacheng Xia, Gaoxiong Zeng, Junxue Zhang, Weiyan Wang, Wei Bai, Junchen Jiang, and Kai Chen. Rethinking transport layer design for distributed machine learning. In *APNet*, pages 22–28. ACM, 2019.

[18] Hao Wang, Jingrong Chen, Xinchen Wan, Han Tian, Jiacheng Xia, Gaoxiong Zeng, Weiyan Wang, Kai Chen, Wei Bai, and Junchen Jiang. Domain-specific communication optimization for distributed DNN training. *CoRR*, abs/2008.08445, 2020.

[19] Shigang Li, Tal Ben-Nun, Salvatore Di Girolamo, Dan Alistarh, and Torsten Hoefler. Taming unbalanced training workloads in deep learning with partial collective operations. In *PPoPP*, 2020.

[20] Xiangru Lian, Ce Zhang, Huan Zhang, Cho-Jui Hsieh, Wei Zhang, and Ji Liu. Can decentralized algorithms outperform centralized algorithms? A case study for decentralized parallel stochastic gradient descent. In *NeurIPS*, 2017.

[21] Mahmoud Assran, Nicolas Loizou, Nicolas Ballas, and Michael G. Rabbat. Stochastic gradient push for distributed deep learning. In *ICML*, 2019.

[22] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over commodity ethernet at scale. In *SIGCOMM*, pages 202–215. ACM, 2016.

[23] Bairen Yi, Jiacheng Xia, Li Chen, and Kai Chen. Towards zero copy dataflows using RDMA. In *SIGCOMM Posters and Demos*. ACM, 2017.

[24] Li Chen, Ge Chen, Justinas Lingys, and Kai Chen. Programmable switch as a parallel computing device. *CoRR*, abs/1803.01491, 2018.

[25] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael M. Swift. ATP: in-network aggregation for multi-tenant learning. In *NSDI*. USENIX Association, 2021.

[26] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter Richtárik. Scaling distributed machine learning with in-network aggregation. In *NSDI*. USENIX Association, 2021.

[27] Mohammad Alizadeh, Albert G. Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *SIGCOMM*. ACM, 2010.

[28] Li Chen, Shuihai Hu, Kai Chen, Haitao Wu, and Danny H. K. Tsang. Towards minimal-delay deadline-driven data center TCP. In *HotNets*, pages 21:1–21:7. ACM, 2013.

[29] Wei Bai, Kai Chen, Hao Wang, Li Chen, Dongsu Han, and Chen Tian. Information-agnostic flow scheduling for commodity data centers. In *NSDI*. USENIX Association, 2015.

[30] Ziyang Li, Wei Bai, Kai Chen, Dongsu Han, Yiming Zhang, Dongsheng Li, and Hongfang Yu. Rate-aware flow scheduling for commodity data center networks. In *INFOCOM*. IEEE, 2017.

[31] Li Chen, Justinas Lingys, Kai Chen, and Feng Liu. Auto: scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In *SIGCOMM*. ACM, 2018.

[32] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with varys. In *SIGCOMM*. ACM, 2014.

[33] Hong Zhang, Li Chen, Bairen Yi, Kai Chen, Mosharaf Chowdhury, and Yanhui Geng. CODA: toward automatically identifying and scheduling coflows in the dark. In *SIGCOMM*. ACM, 2016.

[34] Yangming Zhao, Kai Chen, Wei Bai, Minlan Yu, Chen Tian, Yanhui Geng, Yiming Zhang, Dan Li, and Sheng Wang. Rapier: Integrating routing and scheduling for coflow-aware data center networks. In *INFOCOM*. IEEE, 2015.

[35] Xinchen Wan, Hong Zhang, Hao Wang, Shuihai Hu, Junxue Zhang, and Kai Chen. RAT - resilient allreduce tree for distributed machine learning. In *APNet*. ACM, 2020.

[36] Shriram Sarvotham, Rudolf H. Riedi, and Richard G. Baraniuk. Connection-level analysis and modeling of network traffic. In *SIGCOMM*, 2001.

[37] Wei Dai, Yi Zhou, Nanqing Dong, Hao Zhang, and Eric P. Xing. Toward understanding the impact of staleness in distributed machine learning. In *ICLR*, 2019.

[38] Jianyu Wang, Anit Kumar Sahu, Zhouyi Yang, Gauri Joshi, and Soummya Kar. Matcha: Speeding up decentralized sgd via matching decomposition sampling. In *ICC*. IEEE, 2019.

[39] Pavel Izmailov, Dmitrii Podoprikhin, Timur Garipov, Dmitry P. Vetrov, and Andrew Gordon Wilson. Averaging weights leads to wider optima and better generalization. In *UAI*, 2018.

[40] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. In *ICLR*, 2017.

[41] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *SIAM Rev.*, 2018.

[42] Angelia Nedic, Alex Olshevsky, and Michael G. Rabbat. Network topology and communication-computation tradeoffs in decentralized optimization. *Proc. IEEE*, 2018.

[43] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. In *BMVC*, 2016.

[44] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The cifar-10 dataset. *online: http://www. cs. toronto. edu/kriz/cifar. html*, 55, 2014.

[45] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Fei-Fei Li. Imagenet: A large-scale hierarchical image database. In *(CVPR 2009)*.

[46] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*, 2019.

[47] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100, 000+ questions for machine comprehension of text. In *EMNLP*, 2016.