Published in IEEE Transactions on Network Science and Engineering, 2025. DOI: 10.1109/TNSE.2025.3595747 ©IEEE

Progress and Bandwidth Aware Partial Model Synchronization With Selective Reduce

Shouxi Luo, Zhen Liu, Qing Rao, Ke Li, Huanlai Xing

***Abstract*—By allowing partial instead of all workers to participate in a round of model synchronization, the recent proposal of *partial reduce* provides a promising way to prevent the entire training from being blocked by straggler workers in heterogeneous data-parallel distributed training. However, its current designs are far from optimal, as it selects workers for partial model synchronization agnostic to both their training progress and available bandwidths. To address these issues, we analyze the design space and propose *selective reduce*. By exploring the idea of *waiting* for more workers to be ready and splitting them into partial synchronization groups, based on the state of their training progress and available bandwidths, *selective reduce* could not only enlarge the scale of synchronization (i.e., the number of involved workers) thus accelerating the convergence of the training but also reduce the time cost of synchronization thus making the training iterate faster. Extensive evaluations confirm that *selective reduce* outperforms *partial reduce* and is robust to both inaccurate bandwidth estimations and unknown-in-advance runtime distributions of training computation.**

***Index Terms*—Partial reduce, distributed training, scheduling**

I. Introduction

Nowadays, machine learning, especially deep learning, has empowered vast success services in production, including *advertising recommendation*, *object classification*, *email filtering*, *machine translation*, etc [1], [2], [3], [4], [5]. Nevertheless, with the development of technology and enrichment of training data, new models are continued to be proposed for various purposes like better model accuracy and/or generality [6], higher computational efficiency [7], and lower resource requirements [8], [9]. Accordingly, the efficient training of models plays a key role in the development of new models [10]. Data parallelism techniques have been widely used for this goal [2], [3]: by spreading the massive training data along with the involved computation tasks to a group of workers (e.g., GPU or TPU servers), data-parallel distributed training could shorten the training time by taking advantage of a large number of accelerators like GPUs and TPUs, with the cost of increased traffic loads among workers. More specifically, to guarantee and accelerate the convergence of the trained model, workers participating in data-parallel training are designed to synchronize their local results, like the computed gradients or updated model parameters, periodically during the training [2], [3]. Currently, the *de facto* design of synchronization is to employ the collective communication primitive of *all reduce* (especially the ring-based implementation since it triggers balanced computation and traffic loads among all involved workers [11], [12]). However, as is known, by default, *all reduce* requires that all workers participate in each round of synchronization, suffering from the problem of slow training iteration and low resource utilization because of the ubiquitous straggler workers occurring in distributed training [13], [14].

In large-scale distributed clusters, due to the heterogeneity of the distributed training caused by various factors including the dynamic competition of resources, complicated and hierarchical network structures, and inconsistent and random training workloads [13], some workers might complete a round of training much slower than others, becoming stragglers. These stragglers are unpredictable and would prevent early-completed workers from moving to the next round, leading to slow training iterations and a high waste of resources [1], [14]. For such a problem, the recent proposal of *partial reduce* [14] is a promising and generic solution. Distinguished from the original *all reduce*, which enforces all workers to take part in each round of global synchronization, *partial reduce* provides a relaxed yet controllable partial synchronization semantic to data-parallel distributed training: by allowing a group of workers to launch a synchronization once their total number reaches the pre-defined requirement of p, it prevents the synchronous process from being blocked by stragglers [1], [14], [15]. Then, these straggler workers would take part in another round of *partial reduce* when completing their local training, and there are p ready workers again [14], [15].

Despite promising to deal with stragglers, the current implementation of *partial reduce* is far from optimal in determining the number of workers for each *partial reduce* operation, yielding room for performance improvement [14]. More specifically, in heterogeneous data-parallel distributed training, workers are ready for model synchronization successively; to get rid of straggler workers, *partial reduce* leverages a logically centralized controller and maintains a queue to collect workers that are ready for synchronization. Once there are p workers in the queue, the controller notifies these p workers to launch an *all reduce* operation [14], immediately (hereafter, we define the number of workers involved in a

The preliminary idea of this paper was presented in the Poster session of IEEE ICNP 2022 [DOI: 10.1109/ICNP55882.2022.9940385]. This work was supported in part by NSFC under Grant 62002300, in part by NSFSC under Grant 2025ZNSFSC0489, in part by the Hebei Natural Science Foundation under Grant F2025525008, and in part by the Fundamental Research Funds for the Central Universities under Grant 2682024ZTPY050. *(Corresponding author: Shouxi Luo.)*

Shouxi Luo is with the School of Computing and Artificial Intelligence, Southwest Jiaotong University, Chengdu 611756, China, and also with Tangshan Institute, Southwest Jiaotong University, Tangshan 063000, China.

Zhen Liu is with the School of Computing and Artificial Intelligence, Southwest Jiaotong University, Chengdu 611756, China.

Qing Rao is with Tangshan Institute, Southwest Jiaotong University, Tangshan 063000, China.

Ke Li and Huanlai Xing are with the School of Computing and Artificial Intelligence, Southwest Jiaotong University, Chengdu 611756, China.

partial reduce operation as its scale). Obviously, such a design is agnostic to both each worker's training progress and the available bandwidth, thus having two performance issues, specifically for the synchronization of large models.

On the one hand, recent studies show theoretically and empirically that the larger (synchronization) scales of *partial reduce* tasks have on average, the higher convergence speeds they are likely to achieve [14], [15]. So, for these consecutive ready workers, instead of launching a *partial reduce* task immediately when there are p workers ready, waiting a while might increase the average scale of synchronization (i.e., the number of the involved workers) significantly. On the other hand, given a set of workers, when the training model is large, the completion time of their synchronization tasks (consider the ring-based *all reduce* as an example) is generally dominated by the worker with the smallest available bandwidth; thus, when the available bandwidths of workers are highly skewed, by waiting a while and then grouping workers with similar bandwidth to form a *partial reduce* task, it is possible to reduce the average completion time. Indeed, in such a case, from the view of a *partial reduce* task, via *waiting*, the bottleneck worker could be replaced with an incoming worker. That is to say, *waiting* does not always enlarge the (average) completion time of *partial reduce*, as it could enrich the possible selections of ready workers. Obviously, the key to addressing these two issues is to explore the benefits of *waiting*, based on the state of both workers' training progress and their available bandwidths, i.e., being progress and bandwidth aware. However, due to the phenomenon of stragglers, it is quite challenging to achieve this goal, as i) the exact time when a worker would be ready for synchronization is unable to be known in advance [13], [14]; and ii) the relationship between a *partial reduce* task's completion time and each worker's available bandwidth highly depends on how the synchronization operation is carried out.

In this paper, we propose the scheme of *selective reduce* to overcome the drawbacks of *partial reduce*. As an improvement of the original *partial reduce*, *selective reduce* also works on a logical central controller to collect the status of workers and launch partial synchronization on demand. Given that the iterative training would repeat up to hundreds of thousands of rounds, *selective reduce* uses the runtime distribution of training computation to predict whether training workers would be ready for synchronization in the near future. Based on this and together with the available bandwidth information of workers, *selective reduce* i) adaptively assigns workers with a similar bandwidth to the same group to shorten the time of synchronization and ii) decides whether to wait for more workers, such that the scale of partial synchronization can be enlarged, and/or workers with the bottleneck bandwidth in a partial synchronization group can be replaced for the optimization of completion.

Extensive experiments demonstrate that *selective reduce* could optimize both the scale and the completion time of partial synchronization at the same time, thus bringing benefits to the convergence of the distributed model training. For example, in our heterogeneous test instances, compared with *partial reduce*, *selective reduce* increases the average scale of a round of synchronization up to $1.25\times$ while reducing the average completion time up to $2.55\times$.

To summarize, our main contributions are three-fold.

- A thorough analysis of drawbacks of the original *partial reduce* for heterogeneous distributed training, along with motivating examples showcasing the benefits of progress and bandwidth aware worker selection (Section II).
- *Selective reduce*, a suite of novel adaptive bandwidth and progress aware worker selection algorithms that could optimize both the average scale and the completion time for heterogeneous *partial reduce* operations (Section III).
- Extensive trace-based evaluations confirm the advantages of *selective reduce* over the original *partial reduce* on scheduling heterogeneous distributed training, and show the robustness of *selective reduce* on inaccurate bandwidth estimation and unknown-in-advance runtime distribution of training computation (Section IV).

In the rest of this paper, we first introduce the background and motivation of progress and bandwidth aware worker selection, and summarize the related work in Section II. Then, we propose our design of *selective reduce* in Section III, and evaluate its performance in Section IV. Finally, Section V concludes the paper and discusses the possible future work.

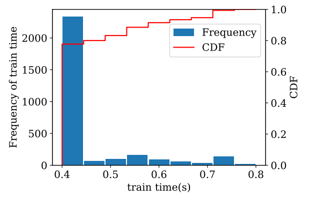
II. Background and Motivation

To motivate the design of *selective reduce*, in this section, we give an overview of the background of data parallelism and *all reduce* in Section II-A, then introduce the problems caused by heterogeneity in Section II-B, and analyze the property of *partial reduce* model synchronization, in Section II-C. After that, we showcase the benefits of bandwidth and progress aware selection for *partial reduce* with motivating examples in Section II-D, and finally discuss related works in Section II-E.

A. Data Parallelism and All Reduce

Nowadays, data parallelism is widely employed to conduct efficient collaborative training of deep neural network models by using a group of selected workers [16], [17]. There are various application scenarios ranging from intra-cluster/datacenter high-performance distributed machine learning (DML) [16], [18], to geo-distributed DML [15], [17], to cross-device federated learning (FL) [19]. In these systems, the training is conducted iteratively. To drive a round, workers first train their replicas of the global model using the locally held datasets in parallel, and then synchronize their results (e.g., gradients or updated model values) via the network [5], [16], [17], [19]. Here, the involved communication task of "aggregating then disseminating workers' results", can be captured by the well-known collective operation of *all reduce*. In some cases, the results of some workers might be more important than those of others. Accordingly, the results would be aggregated in a weighted manner [14]; and we call such a type of *all reduce* as *weighted all reduce* in this paper.

In practice, *all reduce* has various implementations, ranging from peer-to-peer messaging [18], [20] to parameter server (PS) based synchronization [16], [21], [22], [23], [24],


(a) Example of heterogeneity caused by dynamic resource competition

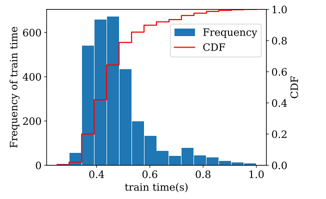
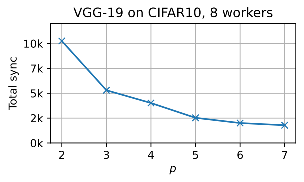

(b) Example of heterogeneity caused by inconsistent training workloads

Fig. 1. Distributions of the runtime of a round of training computation for two typical models observed in real systems and reported by [13]: (a) showcases the result of training the classical ResNet-50 (a CNN model) using a Google Cloud instance equipped with 2 V100 GPUs on dataset ImageNet, and (b) showcases the result of training a Transformer model with P100 GPU on dataset WMT16. The red lines represent the CDF of the computation time of a training computation round.


(a) Total sync

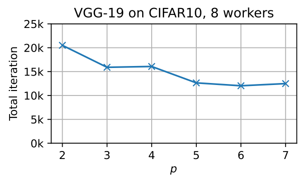

(b) Total iteration

Fig. 2. The impacts of p on the convergence speed of a data-parallel distributed training job built upon partial reduce, reported by work [14].

to algorithm synthesis [25], [26], [27], to tree-based designs [17], [28], to ring-based schemes [12], etc. Basically, different implementations are suitable for various application scenarios and network topologies [29]. For example, ring-based *all reduce* and its variants are widely used in intra-cluster DML, where training workers are generally equipped with high-performance training cards, and hold i.i.d. training datasets [30]. Differently, for cross-device FL, PS-based implementation is popular, where the PS node is also in charge of the dynamic selection and management of training workers [19], [31], [32].

In this paper, we focus on designing schemes to optimize the model synchronization for intra-cluster DML, and argue that our designs can be extended to support other data-parallel training workloads with further efforts.

B. Heterogeneity in Distributed Training

As is known, heterogeneity is abundant in DML, which would cause the problem of stragglers, not only preventing workers from making efficient use of the accelerator resources but also slowing the iteration speed of training down [14]. Recent studies show that heterogeneity is inevitable in production environments. It might stem from i) the dynamic competition of resources like GPUs, I/O capacities, and network bandwidth, ii) complicated and hierarchical network structures, and iii) inconsistent training workloads involved in the training [13], [14]. The runtime distributions of a training computation round for two popular AI models, shown in Figure 1, reported by [13], are typical examples.

Figure 1(a) shows a case of heterogeneity caused by dynamic resource competition. The runtime distribution is obtained by running the classic ResNet-50 training task (a CNN model) on a Google Cloud instance equipped with 2 V100 GPUs, using the IamgeNet dataset [13]. As the computation tasks involved in each training round of ResNet-50 are consistent, the imbalance shown in Figure 1(a) is mainly caused by the performance variability of cloud servers [13]. In contrast, the imbalance of the runtime shown in Figure 1(b), is derived from the various lengths of training data samples used in different rounds. More specifically, the distribution is the result of training a Transformer model on P100 GPU using the WMT16 dataset, which contains a lot of sentences with various lengths, leading to changeable training workloads [13]. Consider that if data parallelism is used to scale up the model training, then, workers who happen to have larger runtimes in a round would act as the unpredictable stragglers.

C. Partial Reduce and Its Property

To relieve the impact of stragglers in data-parallel distributed training, researchers recently proposed the novel design of *partial reduce* [14]. Instead of enforcing all workers (saying n for instances) to participate in each round of synchronization, *partial reduce* allows partial of them to synchronize their locally updated models in a weighted manner, once the number of ready workers reaches the predefined requirement of p ($p \leq n$) [14]. Such a design enables workers who complete their training computations earlier from being blocked by those random stragglers, resulting in more efficient distributed training without breaking the convergence guarantees [14]. Once a straggler node completes the current round of training, it becomes a new ready worker and would participate in the incoming *partial reduce* synchronization. To address the side effects caused by the stale model parameters of the straggler, the reduction computation can be performed with dynamic weights concerning the relative iteration accounts of the involved workers [33].

According to the current design of *partial reduce*, a logical central controller would act as a coordinator to dynamically divide ready workers into groups for partial synchronization. By design, each worker would report to the controller once it completes the local training. And the controller maintains the set of ready workers with a queue and pops them out to launch a weighted *all reduce* collective immediately, each time the length of the queue reaches p. Despite that *partial reduce* can reduce the impact of stragglers, its current implementation is agnostic to both the training progress and available bandwidth of workers, yielding two performance issues for the synchronization of large models.

Specifically, a larger scale of *partial reduce* would generally accelerate the convergence of the data-parallel distributed training [14]. As an example, Figure 2 shows the number of partial reduce synchronization and the total training iterations that eight workers need to perform, to make a VGG-19 model reach the accuracy of 90% on the dataset CIFAR10, under different p settings. The data is directly extracted from the results reported in [14], where more details can be found. We also observe consistent results in our own tests [15]. As the results show, with the growth of p, both the number of needed partial reduce synchronization (Figure 2(a)) and total training iteration (Figure 2(b)) are prone to decrease.

Based on these observations, we now analyze the optimization opportunities for *partial reduce* based training. Let $\pi(p)$ be the total training iteration that workers need to perform to complete a training using *partial reduce* under the setting of p, and t_c and $t_s(p)$ be the average time each worker would take to complete a round of training and a *partial reduce*, respectively. Here, $t_s(p)$ also includes the possible waiting time, and t_c is not a function of p, since the completion of the worker's training computation is generally independent of the value of p. Then, for a distributed training task involving n workers, we could estimate $T(p)$, the completion time of the data-parallel distributed training driven by partial reduce, using E.q. (1).

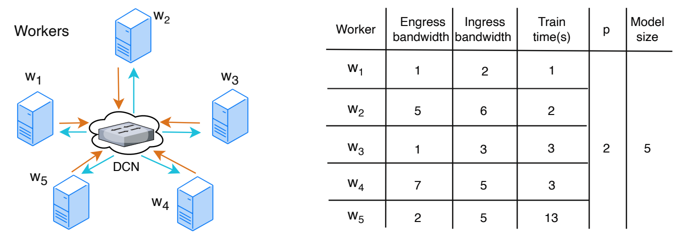
$$T(p) \approx \frac{\pi(p)}{n}(t_c + t_s(p)) \tag{1}$$

Theoretically and practically, the precise value of $\pi(p)$ is jointly determined by a lot of hyperparameter settings [15], [34], [35] in a very complicated way, making it indescribable in formulas. Given that $\pi(p)$ is prone to decrease with the growth of p, if we can optimize the partial reduce synchronization to let $t_s(p)$ stay consistent or even reduce, the entire distributed training can be accelerated. This gives us insights into optimizing the performance of partial reduce model synchronization.

D. Why Bandwidth and Progress Aware Selection

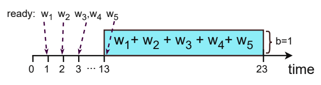
As a larger average scale of partial reduce synchronization (i.e., the number of involved workers) generally leads to higher convergence speeds, when there are p ready workers, instead of launching the collective operation immediately, it is possible to increase the average synchronization scale by waiting a while, which benefits the convergence by decreasing $\pi(p)$. Following this design, if there are more than the minimum required ready workers in the queue, by dividing workers with similar available bandwidth into the same group, the average time cost of a round of synchronization could be reduced, accelerating the iteration of training by shortening $t_s(p)$. We refer to the above design insights as bandwidth and progress aware selections, and now, showcase the benefits with motivating examples.

Consider that five workers labeled from w_1 to w_5 in a cluster are training a model with a size of 5 units. As Figure 3(a) shows, they have various ingress and egress available bandwidth; and due to the heterogeneity in distributed training, they would complete the current round of training at time 1s, 2s, 3s, 3s, and 13s, respectively. Obviously, w_5 acts as a straggler in this round. We assume that workers carry out the synchronization with ring-AllReduce [12]; and the completion time of synchronization for a group of workers can be approximately formulated by E.q.(2), i.e., $2m\alpha + \frac{2v}{b}$. Here, m is the size of the collective group, α is the transmission latency between any workers, v is the size of the model (i.e., 5), and b is the minimum bandwidth among all involved workers. As a

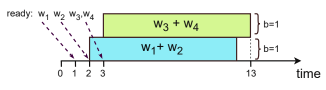


Worker	Engress bandwidth	Ingress bandwidth	Train time(s)	p	Model size
w_1	1	2	1	2	5
w_2	5	6	2		
w_3	1	3	3		
w_4	7	5	3		
w_5	2	5	13		

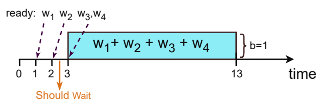
(a) Settings of the motivating example


ready: w_1 w_2 w_3,w_4 w_5
w_1+ w_2 + w_3 + w_4+ w_5
b=1
0 1 2 3 ··· 13 23 time

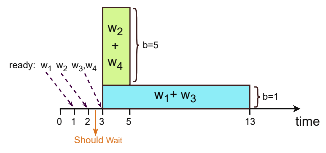
(b) All Reduce



(c) Partial Reduce



(d) Progress-aware selection



(e) Bandwidth-aware selection

Fig. 3. A motivating example: (a) Settings of the example, where workers are connected with a data center network (DCN); (b) All reduce, $avg(CT) = 10, avg(p) = 5$; (c) Partial reduce, $avg(CT) = 10, avg(p) = 2$; (d) Progress-aware selection, $avg(CT) = 10, avg(p) = 4$; (e) Bandwidth-aware selection, $avg(CT) = 6, avg(p) = 2$.

simplified example, we assume that the transmission latency between workers is ultra-low (i.e., $\alpha \ll 1$) thus the time cost of synchronization is dominated by $\frac{2v}{b}$.

Following these definitions, as Figure 3(b) demonstrates, when *all reduce* is used, given the minimum bandwidth is 1, the first four workers, w_1, w_2, w_3, and w_4, would get blocked until w_5 is ready; then, they launch the synchronization at 13s and complete this task at 23s. However, if the scheme for *partial reduce* with $p = 2$ is used, based on the workers' arriving orders, the first four could form two groups $\{w_1, w_2\}$ and $\{w_3, w_4\}$ to start and even complete two separate rounds of synchronization before w_5 is ready, i.e., Figure 3(c), greatly reducing the time that workers are idle.

If the *partial reduce* controller is aware that w_3 and w_4 would be ready at time 3s, then by just waiting for 1s, it could extend the scale of synchronization from 2 to 4, as Figure 3(d) shows. Indeed, alternatively, at the time instance of 3s, by taking the available bandwidth of workers into account, the controller can even divide these four workers into two groups: $\{w_1, w_3\}$ and $\{w_2, w_4\}$. As sketched in Figure 3(e), following this, the average completion time of synchronization can be reduced from 10s to $\frac{10+2}{2} = 6$s. These two examples imply that, by making use of the information of both the training progress and the available bandwidth of workers, it is possible to optimize the scale and per-round completion time of model synchronization for *partial reduce*.

E. Related Work

Optimizing the model synchronization involved in data-parallel DML is a hot research topic and there are abundant related works [2], [3], [36]. In this subsection, we have a broader discussion of these related papers.

TABLE I
THE DIFFERENCES AMONG RELATED STRAGGLER-TOLERANT SYNCHRONIZATION SCHEMES.

# Scheme	Remark	Communication patterns	Bandwidth-aware	Progress-aware
BSP/ASP/SSP [37]	-	Centralized (PS)	✗	✗
DSSP [38]	Based on SSP	Centralized (PS)	✗	✗
DYNSGD [33]	Based on SSP	Centralized (PS)	✗	✗
Sync-Switch [39]	Based on BSP and ASP	Centralized (PS)	✗	✗
D-PSGD [40]	-	Decentralized	✗	✗
AD-PSGD [41]	Relax of D-PSGD	Decentralized	✗	✗
Prague [42]	Based on AD-PSGD	Decentralized	✗	✗
Eager-SGD [13]	Relax of *all reduce*	Decentralized	✗	✗
Partial reduce [14]	Relax of *all reduce*	Decentralized	✗	✗
CREW [15]	Based on *partial reduce*	Decentralized	✓	✗
Selective reduce	Based on *partial reduce*	Decentralized	✓	✓

1) Straggler-Tolerant Synchronization: As pointed out by [13], various factors in training could make workers' completion time for a round of local training skewed, causing the problem of stragglers, and making bulk synchronous parallel (BSP) based synchronization schemes like *all reduce* suffer from performance issues. Besides *partial reduce*, asynchronous communication schemes like asynchronous parallel (ASP), stale synchronous parallel (SSP) [37], D-PSGD [40], AD-PSGD [41], [42], Eager-SGD [13] are alternative proposals. Among them, ASP and SSP are originally implemented using PS; and SSP can be treated as the generalization of ASP and BSP [37]. It guarantees that the gap between workers' training iterations would not exceed the predefined threshold of k. Thus, SSP becomes BSP when $k = 0$ and turns into ASP when $k = +\infty$. Based on SSP, DYNSGD [33] employs a dynamic learning rate schedule scheme to deal with the impacts of stale gradients for performance improvement; and DSSP [38] further designs a scheme to dynamically adjust the staleness thresholds for workers to reduce the waiting time based on the observed iteration time intervals. Different from SSP, DYNSGD, and DSSP, Sync-Switch [39] employs the design of dynamically switching the synchronization setting from BSP to ASP during distributed training.

Different from these PS-based centralized model synchronization designs, schemes like D-PSGD [40], AD-PSGD [41], [42], Eager-SGD [13], and *partial reduce* [14] are the asynchronous relaxations of BSP in the context of decentralized training to tolerate stragglers. D-PSGD [40] shows that, by only synchronizing with its neighbors according to a fixed communication topology, training workers can converge to the final model efficiently. To accelerate the synchronization, AD-PSGD [42] moves a step from D-PSGD to allow workers to just synchronize with a randomly selected neighbor each time, regardless of whether their iteration rounds are the same or not. And Prague [42] further provides an efficient implementation of batched AD-PSGD operations. Differently, when there are straggler workers, Eager-SGD [13] allows these straggler workers to participate in the synchronization with stale model values; and instead, *partial reduce* [14] only launches the synchronization operation for these ready workers, while limiting the minimum number of workers involved in each round of partial synchronization. More recently, the work of CREW [15] also showcases how to achieve efficient *partial reduce* operations for fully-connected training workers by making usage of all available workers and their connections. Indeed, these schemes have various properties and thus are suited for different use cases. A brief comparison of their main differences is summarized in Table I.

2) Topology-Aware Collectives: As different *all reduce* implementations have different traffic patterns, to execute *all reduce* operations efficiently, the characteristics of the underlying network among the training workers (e.g., the topology and the bandwidth) should be taken into account [26], [27], [29]. For this purpose, abundant topology-aware *all reduce* schemes [25], [26], [27], along with various collective communication libraries like NCCL [12], have been proposed.

For example, AutoCCL [43] proposes a search algorithm to tune the performance of NCCL by exploring the impacts of its configuration parameters,such as whether to use a *tree-based* or *ring-based* implementation for *all reduce*. To address the heterogeneity of the connections among the training workers in public clouds, PLINK [44] designs a two-level tree/forest-based hierarchical *all reduce* scheme for model aggregation. In addition, to make efficient use of the heterogeneous link among training workers, BLINK [45] designs algorithms to launch multiple spanning trees for the execution of *all reduce*. To achieve efficient *all reduce* for geo-distributed training workers, MTREE [17] employs similar but improved designs for both the generation and selection of spanning trees, resulting in increased throughput with a reduced number of trees. Moreover, for cases where only parts of the workers participate in the synchronization, STree [28] generates Steiner rather than spanning trees for the *all reduce* operation and formulates the problem as directed Steiner forest packing to optimize. Instead of establishing these trees in advance, NetStorm [46] generates trees with respect to the network status, dynamically, and uses auxiliary paths composed of idle links for acceleration on demand.

Different from using rings or trees, several recent works, such as TECCL [26], TACOS [27], and TARS [25], try to synthesize optimized topology and heterogeneity-aware transmission schemes for *all reduce* by formulating the scheduling problem as math models to solve. And rather than generate communication schemes to match the underlying network topology, several recent works employ the contrast design of reconfiguring the network topology on demand, to support model synchronization schemes built upon parameter servers (e.g., PSscheduler [21]) or Ring-based *all reduce* (e.g., SiP-ML [47], TopoOpt [48] better.

3) Compressed Communication: To mitigate the bottleneck efforts of model synchronization, a promising design is to reduce the involved traffic by schemes like top-k sparsification and gradient quantization. For example, Ok-Topk [49] achieves scalable sparse *all reduce* using top-k sparsification with estimated thresholds; Global-QSGD [50] designs a quantization scheme that can work with tree-based *all reduce* operations seamlessly. As the network status might change during the training, DC2 [51] dynamically adjusts the value of k for top-k sparsification respecting the network congestion; likewise, AQGB [20] designs a truncation-based quantization scheme for gradients, along with a novel algorithm to control the level of quantization with respect to both the network status and training progress. The work of Qsparse-Local-SGD [52] also designs schemes to combine the usages of both sparsification and quantization. In many cases, the involved compression and decompression operations might introduce non-trivial computational overheads, impacting the end-to-end performance of compression-based model synchronization. The recent work of [53] discusses the importance of taking these factors into account when selecting the optimization metric; and Espresso [54] explores the impacts of different compression strategies on end-to-end training performance.

4) In-Network Accelerations: In PS-based model synchronization, with the number of workers increasing, the centralized PS is prone to becoming the performance bottleneck. A lot of recent papers have taken advantage of the capacity of emerging network devices for in-network aggregation, resulting in in-network accelerated *all reduce* [16], [55]. For example, ATP [55] designs schemes to enable P4-programmable switches to act as such accelerators, and SwitchML [56] uses P4-programmable switches to fully replace parameter servers. In addition to P4-programmable hardware, ALEPH [57] and SoftINA [22] also show the possibility of designing softwareized aggregators for in-network acceleration based on the eBPF technique and the virtual switch platform, respectively. THC [58] and AQINA [59] design tensor compression schemes that are compatible with in-network aggregation for joint performance optimization.

Given that in-network aggregation requires aggregatable flows to go through the same aggregator during the journey, solutions like GRID [60], ARO [16], HINA [61], and EINA [24] formulate aggregator-aware routing optimization as various math models and design time-efficient algorithms to solve them. Beyond routing optimization, SPAR [62], PARING [63], and ATRO [23] design algorithms to further jointly optimize the deployment of aggregators, the placement of training jobs, and the topology of the underlying network, respectively to optimize the benefits of in-network aggregation.

As in-network acceleration breaks the one-to-one communication pattern from each worker to the parameter server, the support from the transport protocols is also needed. In this respect, as case studies, A2TP [64] designs schemes to control the sending rate of workers and the allocation of aggregator resources; INP [19], [31] designs new transport protocols involving novel algorithms for progress synchronization, cache allocation, and flow and congestion control for edge-based in-network aggregation; furthermore, MTP [65], [66] designs a novel message-oriented protocol and resource allocation schemes to support more generic in-network computing in modern datacenters.

5) Computation-Communication Overlapping: Last but not least, overlapping the communication with the involved training computation is another powerful design to avoid the communication bottlenecks involved in DDL. For example, ByteScheduler [67] analyzes the opportunity of computation-communication overlapping (CCO) in data-parallel distributed training through fine-grained tensor partition and communication scheduling, and designs a generic framework for this purpose. Instead of splitting the tensors at the level of data volume, DeAR [68] decouples each *all reduce* operation into two sub-operations of *ReduceScatter* and *AllGather* for communication scheduling, reducing the possible startup overhead of communication. PipeDAP [69] further optimizes the execution order of decoupled *ReduceScatter* and *AllGather* operations for performance improvements. In many cases, the communication might not be fully overlapped by the training computation. For these scenarios, the recent work of AQGB [20] designs schemes to conduct CCO-aware adaptive gradient quantization to reduce the amount of exposed communication. Nowadays, intra-layer model parallelism designs are widely used to train large models on memory-limited devices. To achieve computation-communication overlapping in such scenarios, recent work of [70] designs schemes to decouple computation operators to increase the opportunities of overlapping. And differently, TileLink [71] fuses communication and computation kernels to generate compute-communication-overlapped kernels.

In this paper, we follow the design of *partial reduce* [14], and overcome its drawbacks with progress and bandwidth aware worker selections (See Table I). As the first step, our current design is tailored to intra-cluster DML, where training workers prefer to use ring-based implementations for full or partial synchronization. For other DML scenarios, more sophisticated implementations are needed for better performance, which is left as future work.

III. Selective Reduce Algorithms

As analyzed in Section II, the key insights behind our proposal are to enhance the performance of partial reduce operations in a best-effort manner, with selective *waiting* and *grouping*. Based on the state of the training progress and the available bandwidth of workers, we would $i)$ let a *partial reduce* operation wait for more workers to be ready and then $ii)$ split them into partial synchronization groups, such that $i)$ the scale of synchronization could be enlarged for accelerated convergence, and $ii)$ the average time cost of each synchronization would be shortened for faster iteration.

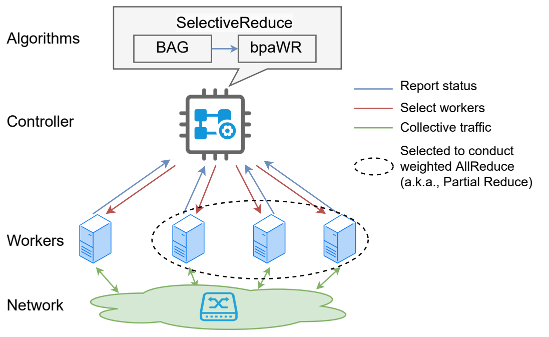


Fig. 4. Our proposed algorithms reside in a logical controller to achieve bandwidth and progress aware selections.

Next, we first present our fundamental assumptions and models (Section III-A), then identify the design challenges (Section III-B). After that, we further propose schemes for bandwidth-aware grouping (BAG) (Section III-C), bandwidth and progress waiting and replacement (BPAWR) (Section III-D), and finally design the joint scheduling algorithm of *selective reduce* as the solution (Section III-E).

A. Assumptions and Models

Distinguished from the raw *partial reduce* [14], the power of *Selective Reduce* mainly stems from two selective designs: i) waiting for more workers to be ready, such that the average scale of synchronization groups could be enlarged, and ii) splitting workers into groups in a bandwidth-aware manner, so that the average communication time of each iteration could be shortened. To be aware of both the training progress and the available bandwidth of workers, as Figure 4 shows, our proposed selective algorithms reside in a logical controller, which collects the time cost of each round of training along with the updated available bandwidth of each worker during the training. Table II summarizes the main notations involved in our algorithm designs. Without loss of generality, we consider that there are n training workers (i.e., $w_1, \cdots, w_n$) networked with a non-blocking data center network, in which bandwidth contentions generally occur at the edge [5], [72], [73]. During a round of synchronization, the size of data a worker would send is generally equal to that it would receive. Thus, for worker w_i, we use b_i to denote its available bandwidth, which is the minimum value among its uplink and downlink bandwidths. As a case study, in this paper, we consider the scenario where the training model is huge and a weighted version of ring-AllReduce is used as the underlying collective primitive for synchronization.

Let α be the link latency between workers, g_i be a group of ready workers selected for synchronization, and β be the time cost of sending one unit of data over the network; for g_i, its $\beta(g_i) \approx \frac{1}{\min_{i:w_i \in g_i} b_i}$. Then, under the assumption that the involved computation is not the bottleneck, the total time of performing ring-AllReduce for a group of workers g_i can be estimated with E.q. (2) [74], [75], where v is the model's

TABLE II
NOTATIONS

Notation	Description
w_i	the unique identifier of the worker i
b_i	w_i's available bandwidth
$B, \{b_i\}$	the set of all workers' available bandwidth
v	the size of the trained model
p	the minimum size of worker groups for *partial reduce*
α	the latency of inter-worker connection
η	a tunable parameter used for bandwidth-aware grouping
θ	a tunable parameter used for progress-aware waiting
Δt	the pre-defined waiting time slot
$P_\Phi(t)$	the probability that a worker computes a round within t
g, g^*, g_Δ	a group of (ready) workers
G	a list of worker groups
L	a list of sorted (ready) workers
$\mathbb{S}$	the set of workers still in training computation
$\mathcal{T}$	the estimated time of performing ring-AllReduce for group g

size.

$$\begin{aligned}\mathcal{T}(g_i) &= 2(|g_i|-1)\alpha + 2\frac{|g_i|-1}{|g_i|}v\beta(g_i)\\ &\approx 2|g_i|\alpha + \frac{2v}{\min_{k:w_k\in g_i} b_k}\end{aligned} \tag{2}$$

When training large models, if the available bandwidth of workers is highly skewed, the optimization room for our algorithm designs is dominated by the selected workers' bandwidth, as E.q. (3) shows.

$$\begin{aligned}&\mathcal{T}(g_i) - \mathcal{T}(g_j)\\ \approx& 2(|g_i|-|g_j|)\alpha + 2v\left(\frac{1}{\min_{k:w_k\in g_i} b_k} - \frac{1}{\min_{k:w_k\in g_j} b_k}\right)\end{aligned} \tag{3}$$

B. Design Challenges

Unfortunately, the two objectives that the selection algorithms would pursue are contradictory, making the design of algorithms challenging. Consider the *selective grouping* of workers as an instance. Suppose that there are $p+k$ workers with various available bandwidths ready for synchronization. For the goal of maximizing the scale of partial synchronization, we should directly select all workers to formulate a $p+k$ group; while for the purpose of minimizing the average time cost, the best scheme is to only select the top-p highest-bandwidth workers to form a group. A similar situation occurs for *selective waiting*: when deciding to wait, should we use the newly arriving workers to enlarge the scale of synchronization, or to replace some low-bandwidth workers such that the average completion time could be reduced?

In the rest of this section, we first describe the design of BAG, a bandwidth-aware grouping algorithm that is able to expand the scale of synchronization with a slightly controlled cost of larger completion times (Section III-C). Then, based on BAG, we further propose BPAWR, a novel algorithm that could optimize both the average scale and completion time for partial synchronization, by conducting bandwidth and progress aware waiting and (worker) replacement (Section III-D). Finally, by putting BAG and BPAWR together, we design SELECTIVEREDUCE, a selectively reduce algorithm

Algorithm 1 BAG: Bandwidth-Aware Grouping

```
1: function BAG($Q, p, \eta, \{b_i\}$)
2:     $L \leftarrow$ sort workers in $Q$ resp. $\{b_i\}$ non-increasingly
3:     $G \leftarrow list()$                          ▷ List of Groups
4:     $g \leftarrow list()$                          ▷ a new group
5:     for each $w_i \in L$ do
6:         if $len(g) \le p$ then
7:             append $w_i$ to $g$
8:             $b^* \leftarrow b_i(1-\eta)$     ▷ $b_i$ is the bandwidth of $w_i$
9:         else if $b_i \ge b^*$ then
10:            append $w_i$ to $g$
11:        else
12:            append $g$ to $G$
13:            $g \leftarrow list()$                  ▷ a new group
14:            append $w_i$ to $g$
15:        end if
16:    end for
17:    if $len(g) > 0$ then
18:        append $g$ to $G$
19:    end if
20:    return $G$
21: end function
```

that achieves optimized and convergence-guaranteed *partial reduce* for heterogeneous data-parallel distributed training (Section III-E).

C. Bandwidth-Aware Grouping

As we have discussed, given $p + k$ ready workers, to maximize the average synchronization scale, the best design is to just select all workers to conduct partial reduce; on the contrary, to minimize the average completion times of synchronization, the best scheme is to sort workers in non-increasing order of their available bandwidth, then slice the first $\lfloor \frac{p+k}{p} \rfloor p$ workers into $\lfloor \frac{p+k}{p} \rfloor$ groups (with the size of p) sequentially. To explore their trade-offs and balance these two optimization goals, we propose the design of η-based bandwidth-aware grouping, i.e., BAG. The insight behind BAG is that the completion time of a round of synchronization for a group of workers is dominated by the one with the smallest bandwidth, saying b_i for instance; then, by including workers with an available bandwidth no smaller than $b_i(1-\eta)$ into the same group, we can expand the scale of synchronization with the controlled maximum time penalty of $\frac{v}{b_i(1-\eta)} - \frac{v}{b_i} = \frac{v}{b_i}\frac{\eta}{1-\eta}$.

Algorithm 1 illustrates details of how the proposed η-based BAG works. Given a queue of currently ready workers Q, the minimum allowed synchronization scale p, the tunable parameter η, and the currently available bandwidth of workers $\{b_i\}$, BAG would split workers in Q into a list of groups $G = [g_1, g_2, \cdots]$. To achieve this, BAG first sorts workers respecting their available bandwidth non-increasingly (Line 2). Then, BAG tries to select p workers to generate a group g and records the relaxed bandwidth threshold b^* (Lines 4-8) for group extension (Lines 9-10). If there does not exist a worker w_i whose available bandwidth meets the requirement of $b_i \ge b^*$, BAG appends the generated group g to G and moves to generate the next group (Lines 11-15). Such a process is repeated until all workers in L have been checked. Finally, the last generated group will be appended as well (Lines 17-19).

Regarding the time complexity, considering that there are n workers in Q. Obviously, the most complex operation of Algorithm 1 is the first step of sorting (Line 2), which can be completed within $O(n\ln(n))$. As the rest of the steps can be done within $O(n)$, the entire Algorithm 1 is $O(n\ln(n))$.

D. Bandwidth- and Progress- Aware Waiting and Replacement

Despite it being hard, if not impossible, to predict the exact completion time of a round of training on a worker, we could get an approximate estimation based on the distributions of the worker's observed runtime. In practice, the training would repeat up to hundreds of thousands of rounds and even more to complete. By formulating the distribution of runtime as a probability model, the controller of *selective reduce* could compute the probability of the instance that a given training worker would complete its current round within the next time slot of $\varDelta t$. Then, based on this observation, we design BPAWR, a bandwidth and progress aware waiting and replacement algorithm to further update the generated worker groups. Such a design makes our proposal work well even without precise knowledge of the probability model at runtime in advance. As evaluations in Section IV-B5 will show, such a design makes our proposal robust to achieve efficient performance without prior knowledge of the distribution.

1) The Probability Model: We use the random variable T to denote the time a worker would take to complete a round of training under the specific parameter settings $\varPhi$ and use $P_\varPhi(T \le t)$ to denote the probability that this worker would complete its current iteration within time t. In practice, the form of $P_\varPhi(T \le t)$ is the same as $F_\varPhi(t)$, i.e., the CDF (Cumulative Distribution Function) of T, which can be approximately computed from the frequency of T observed in the completed iterations. The polylines in Figure 1 show examples. Consider that a worker started a round of training at t time ago and currently, it is still in training (i.e., $P_\varPhi(T > t) = 1$); then, the probability that it would complete the training in the next $\varDelta t$ is $P_\varPhi(T \le t + \varDelta t | T > t)$, which is exactly equal to $F_\varPhi(t)$, the CDF of T, as E.q. (4) shows.

$$\begin{aligned} P_\varPhi(T \le t + \varDelta t | T > t) &= \frac{P_\varPhi(t \le T \le t + \varDelta t)}{P_\varPhi(T > t)} \\ &\overset{P_\varPhi(T>t)=1}{=\!=\!=} P_\varPhi(T \le t + \varDelta t) \\ &= F_\varPhi(t + \varDelta t) \end{aligned} \tag{4}$$

2) Algorithm Details: Algorithm 2 explains the details of how our proposed selective BPAWR algorithm employs the $P_\varPhi(T \le t + \varDelta t | T > t)$-based training progress and available bandwidth information for workers to further optimize the synchronization. Let $\mathbb{S}$ be the set of workers who are still in training and could be waited for, p be the minimum allowed size of a synchronization group, η be the tunable parameter for bandwidth-aware grouping, v be the model size, and $\{b_i\}$ be the currently available bandwidth of each training worker, respectively. Then, given a group of ready workers g, and

Algorithm 2 bpaWR: Bandwidth and Progress aware Waiting and Replacement

```
function BPAWR($\mathbb{S}, g, p, \eta, \Delta t, v, \{b_i\}, \theta$)
    if $len(g) < p$ then
        return False, $\emptyset$, $\mathbb{S}$ ▷ wait for more ready workers
    end if
    $\hat{b} \leftarrow \min_{i:w_i \in g} b_i$
    $C \leftarrow$ PREDICTWORKERS($\mathbb{S}, \Delta t, \hat{b}$)
    $\mathbb{S} \leftarrow \mathbb{S} \setminus \{w_i : (w_i, b_i, p_i) \in C\}$
    $k \leftarrow \lfloor$CALCEXPECTEDNEWWORKERNUM$(C)\rfloor$
    $b \leftarrow$ CALCEXPECTEDBW$(C)$
    $L \leftarrow copy(g)$
    insert $k$ virtual workers whose bandwidth is $b$ into $L$
    $G \leftarrow$ BAG$(L, p, \eta, \{b_i\})$
    $g^* \leftarrow G[0]$
    $g_\Delta \leftarrow set(g) \backslash set(g^*)$ ▷ workers in $g$ but not in $g^*$
    $t_X \leftarrow \mathcal{T}(g) - \mathcal{T}(g^*) \approx \frac{2v}{\min_{i:w_i \in g} b_i} - \frac{2v}{min_{i:w_i \in g^*} b_i}$
    if $t_X > \theta \Delta t$ then
        return True, $g_\Delta, \mathbb{S}$
    else
        return False, $\emptyset, \mathbb{S}$
    end if
end function
function PREDICTWORKERS($\mathbb{S}, \Delta t, b$)
    $C \leftarrow list()$
    for each $(w_i, b_i, t_i) \in \mathbb{S}$ do
        if $b_i > b$ then
            $p_i \leftarrow F_\Phi(t_i + \Delta t)$ ▷ Refer to E.q.(4)
            append $(w_i, b_i, p_i)$ to $C$
        end if
    end for
    return $C$
end function
function CALCEXPECTEDNEWWORKERNUM($C$)
    return $\sum_{(w_i,b_i,p_i) \in C} p_i$
end function
function CALCEXPECTEDBW($C$)
    return $\frac{\sum_{(w_i,b_i,p_i) \in C} p_i b_i}{\sum_{(w_i,b_i,p_i) \in C} p_i}$
end function
```

a pre-defined waiting time slot Δt, BPAWR would decide $i)$ whether a round of synchronization should be triggered immediately, i.e., *waiting* (*True*) or not (*False*); $ii)$ a set of slow-bandwidth workers that should be excluded from g for the optimization of completion time, i.e., $\emptyset$ or g_Δ; and $iii)$ the updated $\mathbb{S}$ that can be used for remaining unprocessed groups.

Basically, if the size of g has not reached the requirement of p, all these workers must wait for more to be ready (Lines 2-4). Otherwise, BPAWR computes $\hat{b}$, the bottleneck bandwidth of workers in g, and filters out the workers in training whose available bandwidth is not less than $\hat{b}$, and computes their probabilities (Lines 6, 22-31) to form a list C. Then, based on C, BPAWR calculates k, the expected number of workers that are likely to be ready within Δt (Lines 8, 32-34), and assumes that these workers are with the bandwidth of b, i.e., the weighted average bandwidth of workers in C (Lines 9, 35-37). By feeding these k expected workers along with these already in g to BAG, BPAWR decides a new grouping plan G for them (Line 12), among which, the first group g^* is the augmented alternative of g, under bandwidth and progress aware waiting and replacement (Line 13). Using this estimated g^*, BPAWR computes both the set of workers that could be excluded from g (and g^*) (Line 14), and the synchronization time that would be saved because of the bandwidth-aware worker replacement (Line 15). Then, if the saved time is larger than θ times of the bound of the waiting time Δ (Line 16), BPAWR would decide to wait and replace (Line 17); otherwise, no waiting and replacement will be conducted for g (Line 19). Here, θ is a tunable parameter, with the default value of 1, to control whether it is worth conducting waiting and replacement. Notable, to avoid these workers being wanted by multiple groups, once workers in C has been checked, BPAWR would remove them from $\mathbb{S}$ (Line 7).

In short, following Algorithm 2, BPAWR evaluates whether it is worth waiting for more workers to be ready based on the status of the available bandwidth and training progress of each worker. As specified by E.q.(4), $F_\Phi(t)$, the distribution of workers' completion times for a round of training (see Figure 1 for examples), provides a way to predict whether a worker would complete its current round within Δt. If this distribution is unknown in advance, BPAWR can directly use the distribution collected by the controller.

As for Algorithm 2's time complexity, no complex loops are involved. Obviously, if there are n workers, each step, including the invoking of functions like PREDICTWORKERS, CALCEXPECTEDNEWWORKERNUM, CALCEXPECTEDBW but BAG (Line 12), could complete within $O(n)$. As BAG is $O(n \ln(n))$, Algorithm 2 is $O(n \ln(n))$.

E. Optimizing Partial Reduce with Selection

Using BAG and BPAWR as building blocks, we design SELECTIVEREDUCE, which could achieve bandwidth and progress aware performance-optimized *partial reduce* for iterative distributed deep training and ensure consistency.

Algorithm 3 shows how the proposed SELECTIVEREDUCE works in the controller: it would repeat until the training should stop (i.e., *ShouldContinueTraining()* returns False, Line 4). Besides parameters $p, \eta, \theta, \Delta t$, and v, it also involves the parameter of c, which specifies that the synchronization must include all workers to conduct a fully weighted *all reduce* after c rounds. To achieve these, SELECTIVEREDUCE uses k to record how many rounds of synchronization have been conducted (Lines 2, 20), which controls the type of triggered synchronization (Lines 5, 8). During the running, SELECTIVEREDUCE uses a queue Q to maintain workers that are ready for synchronization (Line 3), and pop all or part of them to conduct fully (Lines 6-7) or partially (Lines 18-19) weighted *all reduce*, depending on the value of $k\%c$ (Lines 5, 8), respectively. Given that workers have achieved diverse training rounds, thus, like the design of *partial reduce* [14], to achieve better convergence speeds, the synchronization of selective workers is carried out with a weighted variant of *all*

Algorithm 3 Selective Reduce

```
1:  procedure SELECTIVEREDUCE($p, \eta, \theta, \Delta t, v, c$)
2:      $k \leftarrow 0$                         ▷ the current round of training
3:      $Q \leftarrow queue()$                   ▷ Queue of ready workers
4:      while ShouldContinueTraining() do
5:          if $k\%c = 0$ and $len(Q) = n$ then
                                 ▷ perform all reduce every $c$ rounds
6:              $S \leftarrow$ pop $n$ elements from $Q$
7:              launch WeightedAllReduce($S$)        ▷ Async
8:          else if $k\%c \neq 0$ and $len(Q) \geq p$ then
9:              $B \leftarrow$GetUpdatedBandwidthForAllWorkers()
10:             $\mathbb{S} \leftarrow$ GetSetOfWorkersInTraining()
11:             $G \leftarrow$ BAG($Q, \eta, p, \theta, B$)
12:             for each $g_i \in G$ do          ▷ $G : [g_1, \cdots, g_j]$
13:                 $(F, g_\Delta, \mathbb{S}) \leftarrow$ BPAWR($\mathbb{S}, g_i, p, \eta, \Delta t, v, B, \theta$)
14:                 if $g_\Delta \neq \emptyset$ and $i < len(G)$ then
15:                     move workers in $g_\Delta$ from $g_i$ to $g_{i+1}$
16:                 end if
17:                 if $F$ then
18:                     $S \leftarrow$ pop workers in $g_i$ from $Q$
19:                     launch WeightedAllReduce($S$)
20:                     $k \leftarrow k + 1$
21:                 end if
22:             end for
23:         end if
24:         $w \leftarrow$GetReadyWorkerID($\Delta t$)      ▷ $nil$ if timeout
25:         if $w \neq nil$ then
26:             push $w$ to $Q$
27:         end if
28:     end while
29: end procedure
```

reduce. And to support multiple groups of workers performing synchronizations at the same time, SELECTIVEREDUCE launches these *WeightedAllReduce* instances asynchronously. Once an instance of synchronization is completed, the involved workers could continue their training and get marked as *in-training* workers.

On each round of processing, if $k\%c \neq 0$ and the number of ready workers, denoted by $len(Q)$, is larger than p (Line 8), SELECTIVEREDUCE would conduct the selection algorithms to control the execution of partial reduces. To do so, it first obtains the updated bandwidth of each worker (Line 9), along with the set of workers still in training (Line 10). Then, it splits these workers into a list of groups G by using BAG (Line 11). After that, based on the results given by BPAWR (Line 13), it tries to adjust these generated groups (Lines 14-16) and determine whether to launch a round of synchronization (Lines 17-21) for each group in sequence (Line 12), or wait for more workers (Lines 24-27). Here, *GetReadyWorkerID*(Δt) employs a blocking design and would return nil if no new worker becomes ready within Δt.

Obviously, by tuning the value of p, η, θ, and Δt, SELECTIVEREDUCE could balance the two conflicting optimization objectives respecting the requirements of training tasks. Like the original *partial reduce*, a logical controller would run the logic specified in Algorithm 3 based on the observed training progress and bandwidth of workers. As we mainly focus on the case of intra-cluster distributed training in this paper, it is possible to collect the above required status information efficiently via a stand-alone system in practice. Based on them, the controller would invoke BAG and BPAWR to make decisions and launch weighted *all reduce* operations for controlled synchronization. As both BAG and BPAWR are only $O(n \ln(n))$, our proposal scheme would not be the performance bottleneck of the distributed training. Moreover, as the performance evaluation in Section IV-B5 shows, the proposed *selective reduce* is robust to achieve efficient performance even upon inaccurate bandwidth estimations.

IV. PERFORMANCE STUDY

In this section, we evaluate the performance of *selective reduce* through extensive trace-based, event-driven simulations. Detailed results imply that, compared with the original *partial reduce* and *all reduce*, *selective reduce* could improve iteration speed, decrease the synchronization completion time of each round, and expand the scale of synchronization significantly, yielding near-optimal performance.

A. Methodology

1) Network and workloads: We consider that n workers networked with an abstract network switch are training a model collectively. Regarding the available bandwidth of the uplink and downlink that each worker could use for model synchronization (i.e., the value of b_k in E.q. (2)), we assume it is $round(20u, 3)$Gbps, where u follows the uniform distribution of $U[\lambda, 1]$, and by default λ is set to 0.05, yielding a minimum bandwidth of 1Gbps for b_k. For any pair of training workers, we assume its one-way latency is $1ms$ (i.e., the value of α in E.q. (2)). As for the amount of traffic, to highlight the impacts of large models, we assume that models with the size of 500MB. For the time cost of each round of training, we further assume that it is random, but following the observed results of training a Transformer model on dataset WMT16 and training a CNN model on dataset ImageNet, reported by [13], as Figure 1 shows. For the sake of description, we call these two types of training computation time settings *Transformer training trace* and *CNN training trace*, respectively. To test the impacts of training scales, we increase the number of workers n, from 40 to 200 with a step size of 40. And by default, $p = 0.3n$, $\eta = 0.3$ and $\theta = 1$.

2) Simulator, baselines, and metrics: To study the performance of *selective reduce*, we develop a flow-level simulator with Python 3 based on that used in [15], [18], [73]. In brief, at the core of the simulator is a discrete-event simulation engine sharing the similar designs with that of ns3, an open-source network simulator designed for Internet systems. During the simulation, events would be executed one-by-one respecting their logical time, and newly generated events would be inserted into the queue on demand. For the involved communication, link capacities would be allocated to concurrent transfers fairly. By feeding the engine with events configured based on the training traces described in Figure 1, and intentionally controlling the generation of model synchronization events, it could precisely simulate how heterogeneous distributed training systems behave under the control of various synchronization schemes.

Although there are abundant related proposals, the work of [14] has shown that *partial reduce* outperforms existing schemes like ASP, DYNSGD [33], AD-PSGD [41], and Eager-SGD [13], in tests. Among them, DYNSGD is an improvement of the SSP [37], and AD-PSGD is already reported to outperform D-PSGD [40]. As our proposed *selective reduce* is an improvement of the *partial reduce* scheme by design, we could conclude that *selective reduce* is able to outperform these schemes as well. Thus, in tests, we mainly use the *partial reduce* as the baseline. Besides *partial reduce* and *selective reduce*, our simulator also supports the original *all reduce*. For these schemes, we assume that all the involved collective operations are performed on a logical ring.

In practice, the efficiency of distributed training is jointly determined by two factors, i.e., the number of training rounds to reach the stop criteria (a targeted accuracy), and the time cost of each round. As recent studies have shown [14], [15], when triggered *partial reduce* operations have a larger average scale, the training workers could take less rounds of iterations to reach a targeted accuracy. However, there does not exist a simple formula to express the relationship between the average scale of *partial reduce* operations and the actual convergence speed [14], [15]. Despite works like [14] having proved that *partial reduce* has a convergence rate of $O(1/\sqrt{pk})$ under some assumptions, where p and k are the scale and number of synchronization, respectively, such a result bound is very loose. This is because the convergence pattern of a distributed training job is the result of various factors, ranging from the quality of the training dataset, the structure of the model, the settings of the hyper-parameters, etc [15], [18], [34]. To provide generic communication acceleration optimizations for *partial reduce* operations, *selective reduce* tries to shorten the average time cost of each synchronization and enlarge their average scale simultaneously, in the best-effort manner. Accordingly, like recent communication optimization studies [15], [16], [17], [18], we use the following system-related metrics to assess the performance of *selective reduce*.

- **Average sync time**: the average time of a round of synchronization—Smaller values indicate more efficient synchronization (i.e., accelerated iteration);
- **Average sync scale**: the average scale of a round of synchronization—Larger scales generally lead to fewer rounds of iteration for models to converge (i.e., accelerated convergence);
- **Total sync**: the number of synchronization that workers have completed during the training;
- **Total iteration**: the total number of iterations for all workers during the test—A higher value yields better utilization of the computing resources (e.g., GPUs, TPUs);
- **Wasted wait time**: following the plan given by BPAWR, a group of workers might be configured to wait for more ready workers; if no newly ready workers join finally, we call these wait time slots wasted.

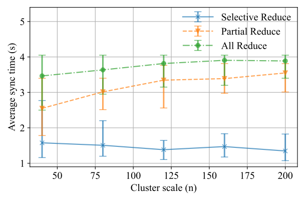
Besides, we also study the influence of tunable parameters like η and θ on the above performance metrics, bandwidth fluctuations and latency. For each parameter setting, we conduct 20 trials to compute and report their $minimum$, $medium$, and $maximum$ values.

B. Performance

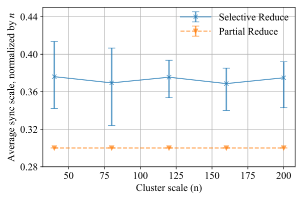
1) Results on the CNN and Transformer training traces: Figures 5 and 6 show the results of *selective reduce*, *partial reduce* and *all reduce* on the two training traces, respectively. By default, in each simulated training instance, workers would stop the training after 100s. As Figures 5(a) and 6(a) show, on both training traces, with the growth of the cluster scale n, the average sync times achieved by *all reduce* and *partial reduce* increase. As a result, for these two schemes, the total amount of training iterations for all workers increase sub-linearly with the growth of the cluster scales (Figures 5(c) and 6(c)), yielding reduced utilization of accelerators like GPU. We argue that this is mainly caused by the joint effects of the increased transmission latency and the heterogeneity of workers' available bandwidth, as specified by E.q.(2). Indubitably, the enlargement of the scale of synchronization could lead to a larger transmission latency for ring-based proposals including *all reduce*, *partial reduce*, and *selective reduce*. Regarding the heterogeneity of bandwidth, however, the mechanism of how it impacts the average sync time of *partial reduce* is a little bit different from that of *partial reduce*. On one hand, for *all reduce*, the probability that there are workers with very low available bandwidth, approximating the floor of $round(20\lambda, 3)$Gbps, would increase with the cluster scales. On the other, for *partial reduce*, given that workers with skewed bandwidth become ready randomly and would be split into groups with the size of p, the probability that there is at least one worker whose available bandwidth is low would increase with the growth of the cluster scale as well, leading to enlarged average sync times. Regarding *selective reduce*, as larger cluster scales, provides a larger room for bandwidth-aware waiting and replacement, the achieved average sync time even decrease.

As shown in Figures 5(b) and 6(b), compared with *partial reduce*, *selective reduce* could improve the average sync scales from the required limit of p to larger values, showing the ability of *selective reduce* on accelerating the coverage of distributed training. Results also show that the cluster size might impact the number of improvements of *selective reduce*, depending on the characteristics of the training workloads. Following these observations and given that *selective reduce* has both optimized average sync time and scales on larger clusters, however, with the growth of n, the total amount of training iterations it achieves larger goes down. We find that this is because workers under the schedule of *selective reduce* would spend more time on waiting, leading to a reduced amount of total sync as Figures 5(d) and 6(d) show.

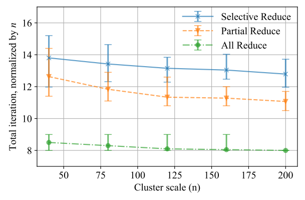
Besides, we also investigate the wasted wait time of each worker under the schedule of *selective reduce*. As Figure 7 reveals, compared with the test duration of 100s, the amount is trivial, only a few milliseconds for each worker, accounting for less than 0.01% of the total training time.



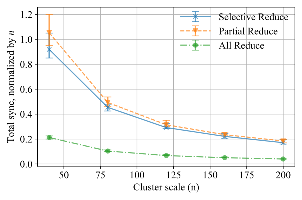
(a) Average sync time



(b) Average sync scale, normalized by n

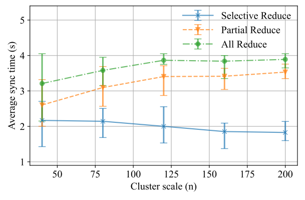


(c) Total iteration, normalized by n

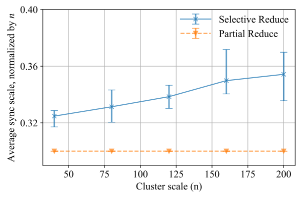


(d) Total sync, normalized by n

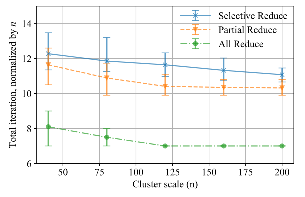
Fig. 5. Results on the CNN training trace show that, compared with *partial reduce*, *selective reduce* is able to launch synchronizations with larger scales and complete them faster, thus achieving more total iteration rounds in the same given time, making better utilization of accelerators.



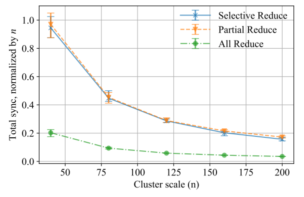
(a) Average sync time



(b) Average sync scale, normalized by n

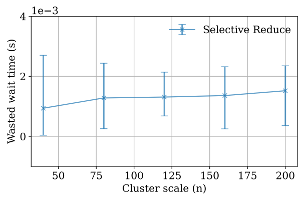


(c) Total iteration, normalized by n

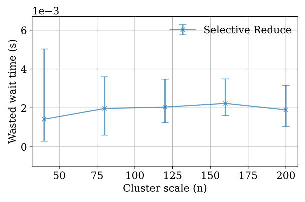


(d) Total sync, normalized by n

Fig. 6. Despite various values being observed, the results on the Transformer training trace show consistent results with those on the CNN training trace, confirming the advantage of *selective reduce* over *partial reduce*.

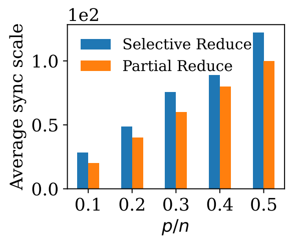


(a) Wasted wait time of *selective reduce* on the CNN training trace

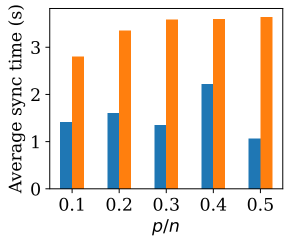


(b) Wasted wait time of *selective reduce* on the Transformer training trace

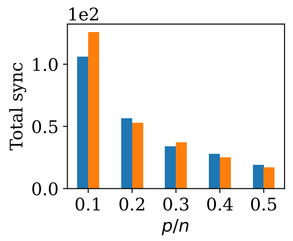
Fig. 7. Compared with *partial reduce* and *all reduce*, the BPAWR algorithm used by *selective reduce* does introduce wasted wait time in some instances; however, as the total amount is tiny, the impact on performance is trivial.



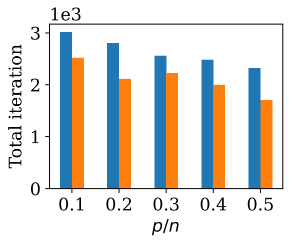
(a) Average sync scale



(b) Average sync time



(c) Total sync



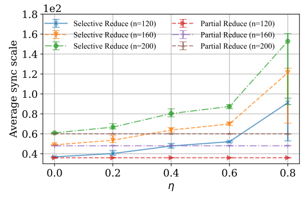
(d) Total iteration

Fig. 8. Impacts of p on the performance of *selective reduce*.

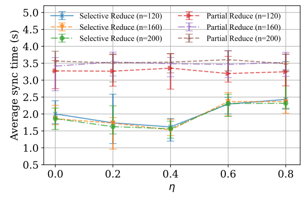
As a summary, results on both traces confirm that the selective designs enable *selective reduce* to outperform *partial reduce* in terms of the average sync time, average sync scale, and total iterations, with improvements up to 1.89-2.55×, 1.19-1.25×, and 1.1-1.17×, respectively. Despite the type of workloads having impacts on the observed result values, results also indicate consistent conclusions. Thus, in the rest of the test, we mainly report the results obtained from the Transformer training trace.

2) Impacts of p: To study the impacts of p on the performance of selective designs, we vary its value from $0.1n$ to $0.5n$ and obtain Figure 8. As expected, for both *selective reduce* and *partial reduce*, with the enlargement of p/n, the average sync scales do grow (Figure 8(a)), at the cost of slowed average sync time (Figure 8(b)) and reduced amount of total sync (Figure 8(c)). Accordingly, the total rounds of training iteration achieved by all workers decrease (Figure 8(d)). It's worth noting that *selective reduce* achieves the smallest average sync time when $p = 0.5n$; this is because in this case, the average sync scale is about 130 (Figure 8(a)), larger than half of the cluster. As a result, the remaining workers in the cluster could never form a group, yielding the smallest amount of total iteration for *selective reduce* (Figure 8(d)). Nevertheless, *selective reduce* always outperforms *partial reduce* in terms of the average sync scale, average sync time, and total iteration in these test instances.

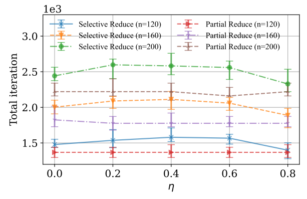
3) Impacts of η: According to the design of Algorithms 1 and 2, the behavior of *selective reduce* is mainly tuned by the setting of η. To investigate the impacts, we increase η from 0 to 0.8, with the step size of 0.2, and obtain Figure 9. Basically, as Figure 9(a) shows, larger η values would lead to larger average sync scales. However, with the increase of η, the average sync time first decreases slightly and then grows



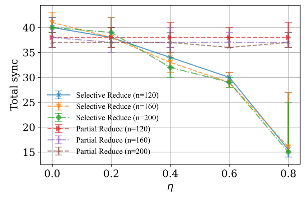
(a) Average sync scale



(b) Average sync time

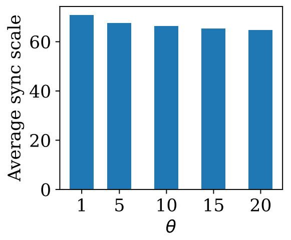


(c) Total iteration

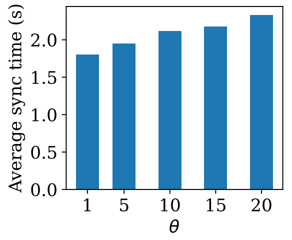


(d) Total sync

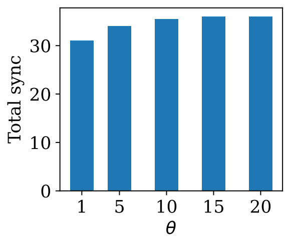
Fig. 9. The impacts of η on the performance of *selective reduce*.



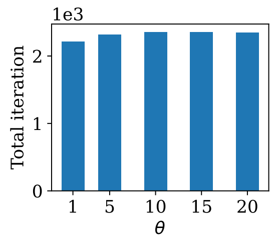
(a) Average sync scale



(b) Average sync time



(c) Total sync



(d) Total iteration

Fig. 10. The impacts of θ on the performance of *selective reduce*.

(Figure 9(b)). We argue that in these test instances when η is not too much, the benefits of the selective replacement design adopted by BPAWR could mask the enlarged sync time caused by the increase of η, and the opportunity of replacement happens to increase with η. Such a result implies that our proposed η-based bandwidth-aware grouping (i.e., BAG) does not necessarily slow the synchronization down. As shown in Figure 9(c), the total iteration conducted by all workers also grows with η at the same time. However, when η is large, e.g., when $\eta = 0.8$, the amount of total iteration decrease. This is because in this case, the average scale of synchronization is larger than $n-p$ (Figure 9(a)). Similar to the case of $p = 0.5n$ shown in Figure 8, the remaining workers could not meet the requirement of p, and the amount of total sync decreases significantly (Figure 9(d)), thus the amount of total iteration is small. Again, in all these instances, *selective reduce* achieves larger and faster synchronizations than *partial reduce*.

4) Impacts of θ: Now, we analyze the impacts of the tunable parameter θ, which is used by BPAWR to decide whether to wait and replace or not. As Figure 10 shows, overall, the performance of *selective reduce* is not very sensitive to θ. Following Algorithm 2, the larger the value θ is, the more conservative waiting and replacement decisions would be made. As a consequence, with θ increasing from 1 to 20, the average sync scale decreases (Figure 10(a)) while the average sync time increases (Figure 10(b)). However, as the total amount of synchronization increases (Figure 10(c)), a slightly larger amount of total iteration is finally obtained (Figure 10(d)).

5) Impacts of unknown training distribution and inaccurate bandwidth estimation: So far, we assume that the controller has knowledge of the distribution of training traces in advance, and can estimate the available bandwidth of each worker precisely. In practice, for a new training task, the distribution of the time cost of a round of training computation might be unknown in advance, and errors might occur in the bandwidth estimation. For these unseen training tasks, the *selective reduce* controller directly uses the distributions observed during the training so far as the trace to estimate the completion of training workers. We call such a scenario as *cold start*; in contrast, the case of in-advance known distribution is named as *warm start*. Results in Figure 11 demonstrate that there is no obvious performance gap between *cold start* and *warm start* for *selective reduce*. To investigate the impacts of inaccurate bandwidth estimation, for a worker with the available bandwidth of b, we artificially introduce an error of $b \times e \times x$ to its estimated values, where x is randomly generated following the uniform distribution of $U[-1, 1]$ and e is a parameter controlling the scale of errors. As Figure 12 shows, in these test instances, for *selective reduce* with both *cold start* and *warm start*, with the increase of the scale of estimation error e, the average sync time slightly increases (Figure 12(a)) and the average sync scale marginally decreases (Figure 12(b)); however, the amount number of synchronization also grows (Figure 12(d)), leading to a little bit more total iterations conducted by workers (Figure 12(c)). Nevertheless, the change in value is not obvious, implying that *selective reduce* is robust to mask these errors and achieve consistent performance.

6) Impacts of bandwidth skewness λ: Recall that in tests, the available bandwidth of workers is generated randomly via $round(20u, 3)$Gbps, where u follows $U[\lambda, 1]$. Accordingly, the smaller λ is, the higher bandwidth skewness workers would have. To study its impacts on the performance of *selective reduce*, we update the value of λ to 0.1 and 0.2 and re-conduct tests. As shown in Figure 13, without surprise, results confirm that the bandwidth-aware selective design makes *selective reduce* always outperforms *partial reduce* in terms of the average sync time, average sync scale, and total iteration; and the more heterogeneous the bandwidth values are, the larger performance gain *selective reduce* could achieve.

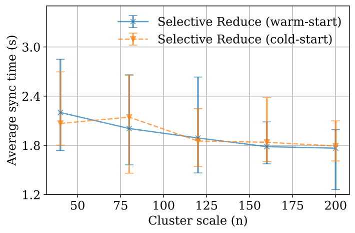
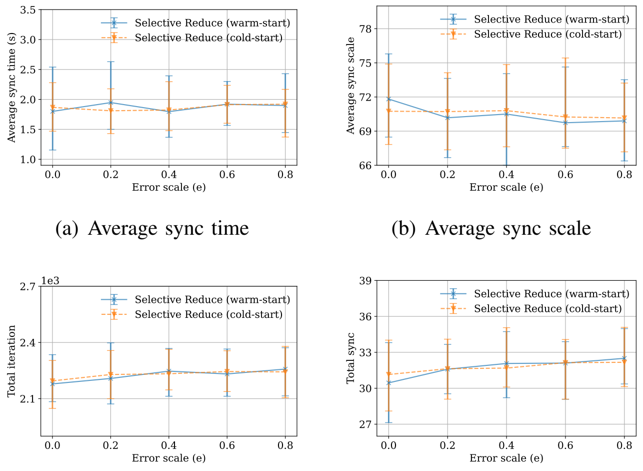


Fig. 11. If the distribution of the time cost of a round of training computation is unknown in advance, by just using the distributions observed during the training (a.k.a, *cold start*), *selective reduce* is able to achieve performances very similar to the case that the distribution is known at the beginning (a.k.a., *warm start*). As the gaps between these two cases are tiny, to be compact, we only show the detailed results of metrics of average sync time here.



(a) Average sync time (b) Average sync scale

(c) Total iteration (d) Total sync

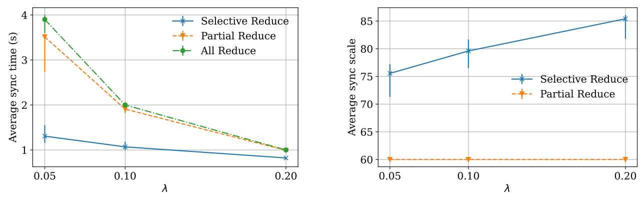
Fig. 12. *Selective reduce* is able to achieve consistent performance when the scale of error increases, implying our proposed algorithm designs are robust.

7) Impacts of latency α: Lastly, we analyze the impact of network delay α on the performance, by varying its value from 0ms to 1ms, to 5ms, and to 50ms. As Figure 14(b) shows, the impacts of α on average sync scale are trivial. However, it is true for other metrics only when α is much smaller than $\frac{v}{b}$, e.g., $\alpha \leq$ 1ms in our tests. Then, consistent with E.q. 2, as shown in Figure 14(a), for all schemes, the average sync time grows with the increase of α, leading to reduced amounts of total iterations (see Figure 14(c)). Also, the growth ratios of *partial reduce* and *selective reduce* are slower than that of *all reduce* since only part of the workers are involved in each round as Figure 14(b) shows. Notably, despite that *selective reduce* has larger average sync scales than *partial reduce*, its average sync time grows more slowly since the impact of increasing α has been partially alleviated by our proposed bandwidth-aware selective design.

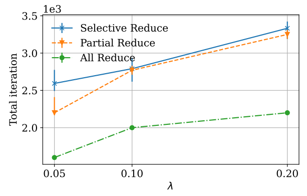
V. Conclusions and Future Work

In conclusion, *partial reduce* is a promising solution for eliminating the impacts of straggler workers in heterogeneous data-parallel distributed training. However, its worker selection scheme is far from optimal, specifically for the synchronization of large models, since it is agnostic to both each worker's training progress and available link capacities. Accordingly, we design *selective reduce*, a suite of adaptive algorithms that could conduct progress and bandwidth aware worker selection to optimize both the scale and completion time of model synchronization for the *partial reduce* solution. Extensive performance studies show the advantage of *selective reduce* over the original *partial reduce* and also indicate that it is robust to unknown-in-advance runtime distributions and inaccurate bandwidth estimation.

Currently, the bandwidth-aware design of the selection algorithms of *selective reduce* is specialized in assuming that synchronization tasks are carried out with ring-AllReduce implementations. As future work, extending *selective reduce* to support other implementations like tree is still open.

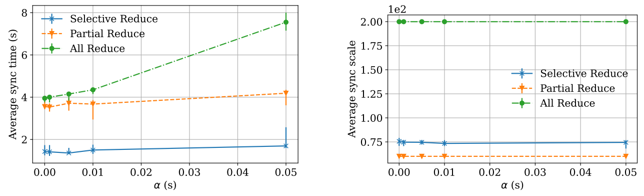


(a) Average sync time (b) Average sync scale

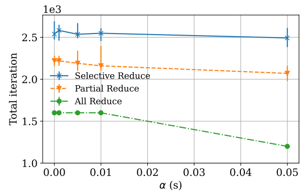


(c) Total iteration

Fig. 13. The impacts of bandwidth skewness λ.



(a) Average sync time (b) Average sync scale



(c) Total iteration

Fig. 14. The impacts of latency α.

References

[1] Z. Liu, S. Luo, K. Li *et al.*, "Poster: Selective reduce for heterogeneous distributed training," in *30th IEEE ICNP*, 2022, pp. 1–2.

[2] J. Verbraeken, M. Wolting, J. Katzy *et al.*, "A survey on distributed machine learning," *ACM Comput. Surv.*, vol. 53, no. 2, mar 2020.

[3] S. Shi, Z. Tang, X. Chu *et al.*, "A quantitative survey of communication optimizations in distributed deep learning," *IEEE Network*, vol. 35, no. 3, pp. 230–237, 2020.

[4] X. Yi, S. Zhang, Z. Luo *et al.*, "Optimizing distributed training deployment in heterogeneous gpu clusters," in *CoNEXT*, 2020, pp. 93–107.

[5] S. Luo, P. Fan, K. Li *et al.*, "Fast parameter synchronization for distributed learning with selective multicast," in *IEEE ICC*, 2022, pp. 4775–4780.

[6] C. Zhou, Q. Li, C. Li *et al.*, "A comprehensive survey on pretrained foundation models: A history from bert to chatgpt," *Int. J. Mach. Learn. & Cyber.*, 2024.

[7] J. Lee, L. Mukhanov, A. S. Molahosseini *et al.*, "Resource-efficient convolutional networks: A survey on model-, arithmetic-, and implementation-level techniques," *ACM Comput. Surv.*, vol. 55, no. 13s, Jul. 2023.

[8] S. Dhar, J. Guo, J. J. Liu *et al.*, "A survey of on-device machine learning: An algorithms and learning theory perspective," *ACM Trans. Internet Things*, vol. 2, no. 3, jul 2021.

[9] J. Zhang and D. Tao, "Empowering things with intelligence: A survey of the progress, challenges, and opportunities in artificial intelligence of things," *IEEE Internet of Things Journal*, vol. 8, no. 10, pp. 7789–7817, 2021.

[10] Y. He, W. Cai, P. Zhou *et al.*, "Beamer: Stage-aware coflow scheduling to accelerate hyper-parameter tuning in deep learning clusters," *IEEE Trans. Netw. Serv. Manag.*, vol. 19, no. 2, pp. 1083–1097, 2022.

[11] A. Sergeev and M. D. Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *CoRR*, vol. abs/1802.05799, 2018.

[12] A. Weingram, Y. Li, H. Qi *et al.*, "xccl: A survey of industry-led collective communication libraries for deep learning," *Journal of Computer Science and Technology*, vol. 38, no. 1, pp. 166–195, Feb 2023.

[13] S. Li, T. Ben-Nun, S. D. Girolamo *et al.*, "Taming unbalanced training workloads in deep learning with partial collective operations," in *PPoPP*, 2020, pp. 45–61.

[14] X. Miao, X. Nie, Y. Shao *et al.*, "Heterogeneity-aware distributed machine learning training via partial reduce," in *SIGMOD*, 2021, pp. 2262–2270.

[15] S. Luo, R. Wang, K. Li *et al.*, "Efficient cross-cloud partial reduce with crew," *IEEE Transactions on Parallel and Distributed Systems*, vol. 35, no. 11, pp. 2224–2238, 2024.

[16] S. Luo, X. Yu, K. Li *et al.*, "Releasing the power of in-network aggregation with aggregator-aware routing optimization," *IEEE/ACM Transactions on Networking*, vol. 32, no. 5, pp. 4488–4502, 2024.

[17] S. Luo, R. Wang, and H. Xing, "Efficient inter-datacenter allreduce with multiple trees," *IEEE Transactions on Network Science and Engineering*, vol. 11, no. 5, pp. 4793–4806, 2024.

[18] S. Luo, P. Fan, K. Li *et al.*, "Efficient parameter synchronization for peer-to-peer distributed learning with selective multicast," *IEEE Transactions on Services Computing*, vol. 18, no. 1, pp. 156–168, 2025.

[19] S. Luo, P. Fan, H. Xing *et al.*, "Eliminating communication bottlenecks in cross-device federated learning with in-network processing at the edge," in *IEEE ICC*, 2022, pp. 4601–4606.

[20] S. Luo, X. Liu, K. Li *et al.*, "Approximate gradient synchronization with adaptive quantized gradient broadcast," *Future Generation Computer Systems*, vol. 174, p. 107983, 2026.

[21] L. Liu, X. Xu, P. Zhou *et al.*, "Psscheduler: A parameter synchronization scheduling algorithm for distributed machine learning in reconfigurable optical networks," *Neurocomputing*, vol. 616, p. 128876, 2025.

[22] X. Yu and S. Luo, "Softina: A softwarized in-network aggregator for distributed applications," in *10th International Conference on Computer and Communications (ICCC)*, 2024, pp. 351–355.

[23] S. Luo, X. Yu, K. Li *et al.*, "Pushing the performance boundary of in-network allreduce with joint topology and routing optimization," in *IEEE/CIC International Conference on Communications in China*, 2025, pp. 1–6.

[24] Z. Qiao, S. Luo, K. Li *et al.*, "Maximizing the throughput of edge-based in-network aggregation with routing optimization," in *IEEE/CIC International Conference on Communications in China*, 2025, pp. 1–6.

[25] W. Lv, S. Luo, K. Li *et al.*, "Towards optimal topology-aware allreduce synthesis," in *IEEE/ACM 32nd IWQoS*, 2024, pp. 1–2.

[26] X. Liu, B. Arzani, S. K. R. Kakarla *et al.*, "Rethinking machine learning collective communication as a multi-commodity flow problem," in *SIGCOMM*, 2024, p. 16–37.

[27] W. Won, M. Elavazhagan, S. Srinivasan *et al.*, "Tacos: Topology-aware collective algorithm synthesizer for distributed machine learning," in *MICRO*, 2024, pp. 856–870.

[28] Z. Zhang, C. Wu, and Z. Li, "Near-optimal topology-adaptive parameter synchronization in distributed dnn training," in *IEEE INFOCOM*, 2021, pp. 1–10.

[29] L. Liu, P. Zhou, G. Sun *et al.*, "Topologies in distributed machine learning: Comprehensive survey, recommendations and future directions," *Neurocomputing*, vol. 567, p. 127007, 2024.

[30] R. Zong, J. Zhang, Z. Tang *et al.*, "Ibing: An efficient interleaved bidirectional ring all-reduce algorithm for gradient synchronization," *ACM Trans. Archit. Code Optim.*, vol. 22, no. 1, Mar. 2025.

[31] S. Luo, P. Zhang, X. Song *et al.*, "Domain-specific transport protocols for in-network processing at the edge: A case study of accelerating model synchronization," *IEEE Transactions on Mobile Computing*, vol. 24, no. 8, pp. 7663–7679, 2025.

[32] L. Luo, C. Zhang, H. Yu *et al.*, "Communication-efficient federated learning with adaptive aggregation for heterogeneous client-edge-cloud network," *IEEE Transactions on Services Computing*, vol. 17, no. 6, pp. 3241–3255, 2024.

[33] J. Jiang, B. Cui, C. Zhang *et al.*, "Heterogeneity-aware distributed parameter servers," in *SIGMOD*, 2017, pp. 463–478.

[34] L. Mai, G. Li, M. Wagenländer *et al.*, "Kungfu: Making training in distributed machine learning adaptive," in *14th OSDI*, Nov. 2020, pp. 937–954.

[35] L. Yang and A. Shami, "On hyperparameter optimization of machine learning algorithms: Theory and practice," *Neurocomputing*, vol. 415, pp. 295–316, 2020.

[36] E. Yu, D. Dong, and X. Liao, "Communication optimization algorithms for distributed deep learning systems: A survey," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 12, pp. 3294–3308, 2023.

[37] Q. Ho, J. Cipar, H. Cui *et al.*, "More effective distributed ml via a stale synchronous parallel parameter server," in *26th NIPS*, 2013, pp. 1223–1231.

[38] X. Zhao, A. An, J. Liu *et al.*, "Dynamic stale synchronous parallel distributed training for deep learning," in *39th ICDCS*. IEEE, 2019, pp. 1507–1517.

[39] S. Li, O. Mangoubi, L. Xu *et al.*, "Sync-switch: Hybrid parameter synchronization for distributed deep learning," in *41st ICDCS*, 2021, pp. 528–538.

[40] X. Lian, C. Zhang, H. Zhang *et al.*, "Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent," in *31st NIPS*, Red Hook, NY, USA, 2017, pp. 5336–5346.

[41] X. Lian, W. Zhang, C. Zhang *et al.*, "Asynchronous decentralized parallel stochastic gradient descent," in *ICML*. PMLR, 2018, pp. 3043–3052.

[42] Q. Luo, J. He, Y. Zhuo *et al.*, "Prague: High-performance heterogeneity-aware asynchronous decentralized training," in *ASPLOS*, 2020, pp. 401–416.

[43] G. Xu, Z. Le, Y. Chen *et al.*, "Autoccl: Automated collective communication tuning for accelerating distributed and parallel DNN training," in *22nd NSDI*. Philadelphia, PA: USENIX Association, Apr. 2025, pp. 667–683.

[44] L. Luo, P. West, J. Nelson *et al.*, "Plink: Discovering and exploiting datacenter network locality for efficient cloud-based distributed training," in *Proceedings of MLSys*, 2020, pp. 82–97.

[45] G. Wang, S. Venkataraman, A. Phanishayee *et al.*, "Blink: Fast and generic collectives for distributed ml," *MLSys*, vol. 2, pp. 172–186, 2020.

[46] Z. Li, W. Feng, W. Cai *et al.*, "Accelerating geo-distributed machine learning with network-aware adaptive tree and auxiliary route," *IEEE/ACM Transactions on Networking*, vol. 32, no. 5, pp. 4238–4253, 2024.

[47] M. Khani, M. Ghobadi, M. Alizadeh *et al.*, "Sip-ml: high-bandwidth optical network interconnects for machine learning training," in *SIGCOMM*. New York, NY, USA: ACM, 2021, pp. 657–675.

[48] W. Wang, M. Khazraee, Z. Zhong *et al.*, "Topoopt: Co-optimizing network topology and parallelization strategy for distributed training jobs," in *20th NSDI*. Boston, MA: USENIX Association, Apr. 2023, pp. 739–767.

[49] S. Li and T. Hoefler, "Near-optimal sparse allreduce for distributed deep learning," in *27th PPoPP*. New York, NY, USA: ACM, 2022, pp. 135–149.

[50] J. Xin, M. Canini, P. Richtárik *et al.*, "Global-qsgd: Allreduce-compatible quantization for distributed learning with theoretical guarantees," in *EuroMLSys*. New York, NY, USA: ACM, 2025, pp. 216–229.

[51] A. M. Abdelmoniem and M. Canini, "Dc2: Delay-aware compression control for distributed machine learning," in *IEEE INFOCOM*, 2021, pp. 1–10.

[52] D. Basu, D. Data, C. Karakus *et al.*, "Qsparse-local-sgd: Distributed sgd with quantization, sparsification, and local computations," *IEEE Journal on Selected Areas in Information Theory*, vol. 1, no. 1, pp. 217–226, 2020.

[53] W. Han, S. Vargaftik, M. Mitzenmacher *et al.*, "Beyond throughput and compression ratios: Towards high end-to-end utility of gradient compression," in *23rd HotNets*. New York, NY, USA: ACM, 2024, p. 186–194.

[54] Z. Wang, H. Lin, Y. Zhu *et al.*, "Hi-speed dnn training with espresso: Unleashing the full potential of gradient compression with near-optimal usage strategies," in *18th EuroSys*. New York, NY, USA: ACM, 2023, pp. 867–882.

[55] C. Lao, Y. Le, K. Mahajan *et al.*, "Atp: In-network aggregation for multi-tenant learning," in *18th NSDI*. USENIX Association, Apr. 2021, pp. 741–761.

[56] A. Sapio, M. Canini, C.-Y. Ho *et al.*, "Scaling distributed machine learning with In-Network aggregation," in *18th NSDI*, Apr. 2021, pp. 785–808.

[57] P. Yang, H. Xu, G. Zhao *et al.*, "Aleph: Accelerating distributed training with ebpf-based hierarchical gradient aggregation," *IEEE/ACM Transactions on Networking*, vol. 32, no. 5, pp. 4128–4143, 2024.

[58] M. Li, R. B. Basat, S. Vargaftik *et al.*, "Thc: Accelerating distributed deep learning using tensor homomorphic compression," in *21st NSDI*. Santa Clara, CA: USENIX Association, Apr. 2024, pp. 1191–1211.

[59] Z. Su, S. Luo, K. Li *et al.*, "Efficient in-network aggregation with adaptive quantization," in *Proceedings of the 9th Asia-Pacific Workshop on Networking*, ser. APNET '25. New York, NY, USA: Association for Computing Machinery, 2025, pp. 247–248.

[60] J. Fang, G. Zhao, H. Xu *et al.*, "Grid: Gradient routing with in-network aggregation for distributed training," *IEEE/ACM Transactions on Networking*, vol. 31, no. 5, pp. 2267–2280, 2023.

[61] H. Zhu and Z. Guo, "Revisiting the in-network aggregation in distributed machine learning," *IEEE Transactions on Networking*, pp. 1–15, 2025.

[62] L. Luo, S. Yang, H. Wu *et al.*, "Maximizing aggregation throughput for distributed training with constrained in-network computing," in *IEEE ICC*, 2023, pp. 3652–3657.

[63] Y. Qiu, G. Zhao, H. Xu *et al.*, "Paring: Joint task placement and routing for distributed training with in-network aggregation," *IEEE/ACM Transactions on Networking*, vol. 32, no. 5, pp. 4317–4332, 2024.

[64] Z. Li, J. Huang, Y. Li *et al.*, "A2tp: Aggregator-aware in-network aggregation for multi-tenant learning," in *18th EuroSys*. New York, NY, USA: ACM, 2023, pp. 639–653.

[65] B. E. Stephens, D. Grassi, H. Almasi *et al.*, "Tcp is harmful to in-network computing: Designing a message transport protocol (mtp)," in *20th HotNets*. New York, NY, USA: ACM, 2021, pp. 61–68.

[66] T. Ji, R. Vardekar, B. Vamanan *et al.*, "Mtp: Transport for In-Network computing," in *22nd NSDI*. Philadelphia, PA: USENIX Association, Apr. 2025, pp. 959–977.

[67] Y. Peng, Y. Zhu, Y. Chen *et al.*, "A generic communication scheduler for distributed dnn training acceleration," in *27th SOSP*. New York, NY, USA: ACM, 2019, pp. 16–29.

[68] L. Zhang, S. Shi, X. Chu *et al.*, "Dear: Accelerating distributed deep learning with fine-grained all-reduce pipelining," in *43rd ICDCS*, 2023, pp. 142–153.

[69] Y. Gao, B. Hu, M. B. Mashhadi *et al.*, "Pipedap: An efficient communication framework for scheduling decoupled all-reduce primitives in distributed dnn training," *IEEE Transactions on Emerging Topics in Computing*, pp. 1–16, 2025.

[70] S. Wang, J. Wei, A. Sabne *et al.*, "Overlap communication with dependent computation via decomposition in large deep learning models," in *28th ASPLOS*. New York, NY, USA: ACM, 2022, pp. 93–106.

[71] S. Zheng, J. Fang, X. Zheng *et al.*, "Tilelink: Generating efficient compute-communication overlapping kernels using tile-centric primitives," in *Eighth Conference on Machine Learning and Systems*, 2025.

[72] S. Luo, H. Yu, K. Li *et al.*, "Efficient file dissemination in data center networks with priority-based adaptive multicast," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 6, pp. 1161–1175, 2020.

[73] S. Luo, P. Fan, H. Xing *et al.*, "Meeting coflow deadlines in data center networks with policy-based selective completion," *IEEE/ACM Transactions on Networking*, vol. 31, no. 1, pp. 178–191, 2023.

[74] J. Fei, C.-Y. Ho, A. N. Sahu *et al.*, "Efficient sparse collective communication and its application to accelerate distributed deep learning," in *SIGCOMM*, 2021, pp. 676–691.

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