Published in IEEE Transactions on Parallel and Distributed Systems (Volume: 35, Issue: 11, November 2024)

Efficient Cross-Cloud Partial Reduce With CREW

Shouxi Luo, Renyi Wang, Ke Li, Huanlai Xing

***Abstract*—By allowing p out of n workers to conduct *all reduce* operations among them for a round of synchronization, *partial reduce*, a promising partially-asynchronous variant of *all reduce*, has shown its power in alleviating the impacts of stragglers for iterative distributed machine learning (DML). Current *partial reduce* solutions are mainly designed for intra-cluster DML, in which workers are networked with high-bandwidth LAN links. Yet no prior work has looked into the problem of how to achieve efficient *partial reduce* for cross-cloud DML, where inter-worker connections are with scarcely-available capacities. To fill the gap, in this paper, we propose CREW, a flexible and efficient implementation of *partial reduce* for cross-cloud DML. At the high level, CREW is built upon the novel design of employing all active workers along with their internal connection capacities to execute the involved communication and computation tasks; and at the low level, CREW employs a suite of algorithms to distribute the tasks among workers in a load-balanced way, and deals with possible outages of workers/connections, and bandwidth contention. Detailed performance studies confirm that, CREW not only shortens the execution of each *partial reduce* operation, outperforming existing communication schemes such as PS, Ring, TOPOADOPT, and BLINK greatly, but also significantly accelerates the training of large models, up to 15× and 9×, respectively, when compared with the all-to-all direct communication scheme and *original partial reduce* design.**

***Index Terms*—Partial reduce, flow scheduling, cloud computing**

I. INTRODUCTION

In today's artificial intelligence (AI) powered world, geo-distributed machine learning (Geo-DML) systems, which provide the ability to learn models from massive data across the globe, have become essential infrastructure for the design, development, and deployment of large-scale AI services [1–3]. For these systems, recent studies show that performing model synchronizations across wide-area networks (WANs) could dominate the time cost of the entire training; thus, optimizing the involved communication becomes the key to improving the efficiency of Geo-DML training [3].

Similar to the case of intra-datacenter distributed training, the synchronization of models involved in Geo-DML can be carried out using the collective operations of *all-* or *partial-reduce*, depending on the training algorithm designs [4].

This work was supported in part by NSFC under Projects 62002300 and 62202392, in part by the Sichuan Science and Technology Program under Project 2023ZHJY0009, in part by the Fundamental Research Funds for the Central Universities under Projects 2682024ZTPY050 and 2682022ZTPY089. A brief description of the preliminary design of this paper was presented in the Poster session of APNet 2022 as a 2-page extended abstract with the title of "Efficient Partial Reduce Across Clouds" [1] [DOI: 10.1145/3542637.3543707]. (Corresponding author: Huanlai Xing.)

The authors are with the School of Computing and Artificial Intelligence, Southwest Jiaotong University, Chengdu 611756, China, and also with the Engineering Research Center of Sustainable Urban Intelligent Transportation, Ministry of Education, Chengdu 611756, China (e-mail: sxluo@swjtu.edu.cn; renyiwang@my.swjtu.edu.cn; keli@swjtu.edu.cn; hxx@swjtu.edu.cn).

Partial reduce is a recently proposed variant of *all reduce* which supports partially-asynchronous training: by allowing p out of n workers to conduct *all reduce* operations among them for a round of synchronization, it is able to ease the impacts of stragglers with a slowed-yet-controllable convergence speed, thus promising for data-parallel distributed training in heterogeneous environments [4]. Several recent works have studied how to achieve efficient *partial reduce* in the context of intra-datacenter distributed training [4–7]. However, none of them has looked into the problem of applying *partial reduce* for Geo-DML. Then, an interesting and important question is: *how could we achieve efficient partial reduce for data-parallel distributed training in the context of Geo-DML, where workers are networked with heterogeneous WAN connections?*

Basically, based on whether the hosting servers of workers belong to the same cloud provider or not, modern Geo-DML applications can be roughly classified into two types, namely intra-cloud and inter-cloud (or cross-cloud, alternatively) Geo-DMLs, respectively. In the former case, training workers are located at multiple geo-distributed datacenters belonging to the same owner; accordingly, the inter-datacenter WANs are likely visible or manageable to the cloud owner and follow specific topologies. Several recent works have reported such applications [8–11]. And in the latter case, hosting servers are generally spread over multiple clouds, among which the WAN is likely to be operated by independent Internet Service Providers (ISP), out of the control of endpoints. Typical examples include that: 1) an enterprise has hosted its data and applications on multiple clouds respecting various purposes, thus its distributed data processing and machine learning tasks are conducted across multiple clouds [12]; 2) different companies or organizations hosted on various clouds cooperatively train the same model with the technique of cross-silo federated learning, a specific type of Geo-DML [13].

Following the latter case, in this paper, we focus on the goal of achieving flexible and efficient *partial reduce* for cross-cloud Geo-DML, where the inter-worker overlay network can be abstracted out as a complete graph (i.e., full-mesh). Despite this full-mesh structure of the inter-worker topology could greatly simplify the relationship between workers, designing an efficient and flexible solution is still quite challenging, due to the following facts. Firstly, just like the case of intra-cloud Geo-DML [2, 10], inter-worker connections in cross-cloud Geo-DML are with scarce and heterogeneous available capacities. Moreover, because of the possible outage of WANs [14, 15], some workers or their connections might be unavailable for a while during the training, making 1) the execution of ongoing cross-cloud *partial reduce* failure, and 2) the inter-worker topology incomplete over a period of time.

To address the above challenges, we propose CREW, a flexible and efficient *partial reduce* implementation that builds

upon the design of "weighted s**C**atter, controlled **RE**duce computation, and **W**ay-back multicast" to pursue the goal. Compared with existing solutions, the power of CREW stems from its novel workflow designs: as the involved computation of reduce is generally not computationally intensive, instead of employing only the p selected workers, CREW leverages all the n available workers to conduct the operations needed by a *partial reduce* in a bandwidth-aware way (via "scatter⇒reduce⇒multicast"; see §III for details), thus can make efficient usage of all the abundant inter-cloud WAN connections. Moreover, to deal with the possible outages of workers and WAN connections during the training, CREW employs novel algorithms to dynamically compute substitute execution plans to bypass these unavailable workers or broken connections. Detailed performance studies confirm that CREW outperforms existing solutions and is able to deal with possible link or node outages, properly (workers and nodes are used interchangeably in this paper).

In summary, we mainly make three contributions:

- CREW, an execution workflow that is able to achieve efficient *partial reduce* for cross-cloud distributed training by making efficient usage of all available workers and their inter-cloud WAN connections (§III).
- A suite of algorithms that enable CREW to execute *partial reduce* tasks in a bandwidth-aware way, robust to worker/connection outages, and efficient for the schedule of concurrent triggered flows, during the training (§IV).
- Extensive studies showcase the efficiency and effectiveness of CREW. It could accelerate the execution of a single *partial reduce* operation up to dozens of times compared to existing communication schemes like PS, Ring, TopoAdopt, and BLINK, and achieve up to $12-15\times$ and $8-9\times$ speedups compared to the all-to-all direct communication scheme and *original partial reduce* design, respectively, for iterative model training (§V).

Aims of CREW. As one type of hyper-parameter, recent studies have shown that, a smaller value of p for *partial reduce* enables the training to iterate faster in the existence of stragglers, with the cost of slowed convergence speed [4]. As the end-to-end training performance (e.g., the total time the trained model takes to reach a targeted test accuracy) depends both on the number of training rounds and the time costs of each iteration, in practice, the best value of p varies across workloads and automatic hyper-parameter tuning is needed [4, 16]. Motivated by these facts, the focus of this paper is to design a generic and robust execution scheme (i.e., CREW) that could provide cross-cloud training workers the ability to achieve *partial reduce* more efficiently. Essentially, the power of CREW stems from the design of letting workers distribute the involved communication and computations among all active workers in a bandwidth-aware manner. For small models, the time cost of a *partial reduce* operation is dominated by the latency of WAN links thus the benefits of CREW are limited and trivial in that case; to be fast, workers can disseminate their local updates to all others via direct all-to-all communication, or by using the controller of *partial reduce* as the parameter server [17, 18]. However, nowadays, large-scale models are becoming more and more popular: deep learning models could involve up to hundreds of billions of parameters [17, 19], yielding bulk transfers for the synchronization of their model updates over WAN links. As a result, the wall time needed for model synchronization would increase and the significance of communication optimization also increases. CREW is mainly designed for these scenarios.

The rest of this paper is organized as follows. We first overview the related background and motivation in Section II, then sketch the design of CREW in Section III. After that, the detailed algorithm designs and performance studies follow in Sections IV and V, respectively. Finally, we discuss related works in Section VI and conclude the paper in Section VII.

II. Background and Motivation

Before looking into the details of algorithm designs, in this section, we first overview the backgrounds of cross-cloud DML (§II-A) and *partial reduce* (§II-B), which motivate the design of CREW, then discuss the corresponding design challenges (§II-C).

A. Cross-Cloud DML

At the early stage of the development of machine learning, the training data is generally collected into a central training cluster located in a private or public cloud for model training. And to accelerate the training, various distributed training frameworks like Parameter Server (PS) [17, 18, 20], Adam [21], Ako [22], Horovod [23], Malt [24], DDP [25], together with abundant parallelism schemes [4, 7, 17, 18, 23, 26, 27], are proposed. In these systems, the training workers generally have homogeneous capacities and are networked with high-bandwidth intra-datacenter links; thus, such a type of training can be named intra-datacenter distributed machine learning. Regarding that large quantities of data are generated in a geo-distributed manner and are prohibited from centralized in many cases, across/inter-datacenter DML has become inevitable [2, 28]. Compared with intra-datacenter distributed training, the connections between workers in the context of inter-datacenter training are WAN links with scarce and expensive capacities. To deal with this, a new form of geo-distributed distributed machine learning is proposed. For example, Cano et al [28] systematically study and present the results of logistic regression models to demonstrate the effectiveness of a geo-distributed approach combined with communication-parsimonious algorithms. Gaia [2] dynamically eliminates insignificant and unnecessary model update information to efficiently utilize the scarce and heterogeneous WAN bandwidth, without specializing the ML algorithms.

Indeed, based on whether the involved datacenters are owned by the same cloud provider or not, geo-distributed inter-datacenter DML can be roughly classified into two categories, namely intra-cloud and inter-cloud (or cross-cloud), respectively. Following this, the aforementioned intra-datacenter DML belongs to the intra-cloud case as well; [8–11] are examples. And the typical cross-silo federated learning [13] and multi-cloud machine learning [12] are typical inter-cloud DML examples. As reported by [29, 30], cross-silo federated

learning has actual applications across various domains like *smart healthcare*, *smart retail*, *credit risk control*, and *anti-money laundering*. In this paper, we focus on optimizing the data transmissions involved in cross-cloud DML.

B. Partial Reduce

The synchronization mechanism of *all reduce* is very popular in modern DML, which has various implementations in practice, including the naive direct peer-to-peer broadcast (i.e., all-to-all), *rings* (e.g., BaiduRing [31], Horovod [23]), *tree*s (e.g., Two-tree [32], MultiTree [33], MTREE [34]), and others (e.g., Butterfly Mixing [35], Recursive Halving and Doubling [36]). In these *all reduce* schemes, all workers are forced to participate in each round of model synchronization, leading to two performance issues. Firstly, they might yield non-trivial burst traffic loads; taking the peer-to-peer broadcast based schemes as an example, n workers would trigger as much as $O(n^2)$ messages to conduct a round of *all reduce*. Secondly, as all workers are involved in each round, they are unable to deal with possible stragglers properly, resulting in significantly-downgraded performance in that case. Fortunately, recent studies show both theoretically and empirically that, a lot of iterative DML algorithms are able to tolerate incomplete model synchronization, leading to a novel generalization of *all reduce*, namely *partial reduce*. By allowing only a portion rather than all workers to conduct a synchronization, *partial reduce* could not only reduce the amount of traffic but also tolerate possible stragglers, with controllable relaxed synchronization for distributed training [4–6, 24, 26, 27, 37].

Regarding the level of consistency, current *partial reduce* schemes can be roughly classified into three types. 1) Different parts of the aggregated model on a worker are computed from (updated by) different sets of training datasets and participating workers would get distinct (mixed) model versions (e.g., Combo [6]). 2) Different parts of the aggregated model on a worker are computed from (updated by) the same set of training datasets, but different participating workers still hold diverse model versions (e.g., Malt [24], Orpheus [5], SelMcast [26], EagerSGD [7]). 3) Different parts of the aggregated model on a worker are computed from (updated by) the same set of training datasets, and workers participating in the same *partial reduce* to obtain the same model after synchronization (e.g., Prague [27], P-Reduce [4]). Basically, the first two types of designs achieve weak consistency, with the possible benefits that involved traffic would be reduced and balanced among workers, and asynchronous communication is allowed. While the third one achieves the strongest consistency, to do so, a group of workers must synchronize their model parameters or gradients completely and accurately, in a synchronous manner. Despite that Combo [6] has proven the convergence ability of convex models under weak-consistent partial reduce, it is still unknown whether this ability is still kept or not for the abundant non-convex deep neural network models. In practice, the quality of a deep model is determined by a lot of hyper-parameters; thus, trial-and-error approaches are widely employed in the development of new models [16]. To limit the possible side effect of *partial reduce* on the distributed training, we argue that strong consistency is necessary.

Currently, there are two typical solutions following the third type of consistency design, namely Prague [27] and P-Reduce [4], powered by two distinct design decisions. On one hand, in Prague, workers are split into *partial reduce* groups without regard to their training states; hence, once there is a straggler in a group, all other early completed workers in the same group would get blocked by it. P-Reduce overcomes this issue by explicitly (only) selecting workers that are already ready to join *partial reduce* operations. On the other, Prague provides no control on the scale of partial reduce, while in P-Reduce, there is a tunable parameter for this purpose (saying p for instance). Recent studies have already shown that, larger *partial reduce* scales generally yield fewer rounds of training to converge, at the cost of increased time cost for each round [4]. For the optimization of the total run time, the best value of p depends on both the training workloads and network status. Thus, to provide a flexible yet convergence-guaranteed *partial reduce* service for inter-cloud DML tasks, CREW follows the design of P-Reduce [4] and provides strong consistency for Geo-DML.

C. Design Challenges

To achieve efficient *partial reduce* for cross-cloud data-parallel DML, the following challenges must be addressed.

Abundant bandwidth-limited WAN connections. First of all, in the context of cross-cloud training, the connections between different workers are generally with limited available bandwidth. Nevertheless, the capacities of different connections are generally independent, and thus the network between workers can be abstracted out as a full mesh. To achieve fast *partial reduce*, the proposed scheme must be able to make use of all the available connections, despite that each of them is only with limited available bandwidth.

Task-aware flow scheduling. Secondly, when there is more than one flow from a worker to the same another one, they will compete for the available bandwidth following the default principle of per-flow fairness. However, a lot of prior studies have shown that such a default fair-sharing bandwidth allocation at the level of per-flow generally leads to performance far from optimal at the level of task (or entire application), and task-aware flow scheduling is a promising fix [38]. Thus, a flow scheduling scheme that could achieve efficient flows respecting the goal of optimizing the progress of *partial reduce* is needed by the solution.

Network outage. Last but not least, due to various reasons [14, 15], the inter-cloud WANs might suffer from network outages, making a worker unavailable to some or all other workers for a certain period during the training. Such an issue would lead to node or link failures during the model synchronization and is more likely to happen when the scale of training is large. To be practical, the proposed solution must be able to deal with these issues properly.

III. CREW OVERVIEW

CREW provides *partial reduce* semantics to workers by leveraging all active workers along with their available WAN

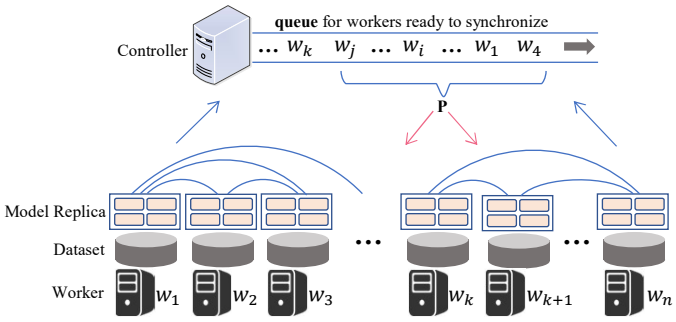


Fig. 1: CREW Architecture.

connections to execute the needed communication and computation operations for each *partial reduce* task. To conduct consistent and bandwidth-aware *partial reduce*, as Figure 1 shows, CREW involves a logical controller, which maintains a global view of the entire network and could make smart decisions. Basically, workers establish long-lived connections with the controller to report the states of their WAN connections and training progress (e.g., being ready for *partial reduce*). Based on these, the controller decides which group of workers to execute a *partial reduce* along with what the best execution plan is, and replies. In practice, workers are ready for synchronization one after another; given a configurable parameter of p, like [4], the CREW controller maintains a queue to record which workers are ready for *partial reduce*, and would pop them out (denoted by **P**) to do so once the number of workers in the queue reaches p. Once any workers report the loss of connections to other workers, the controller also recomputes the execution scheme to avoid the usage of these broken connections or unavailable workers, and notifies the new scheme to involved workers immediately.

As Figure 2 shows, the workflow to execute a *partial reduce* mainly involves three phases, namely *weighted scatter*, *controlled reduce computation*, and *way-back multicast*, respectively. Consider that the training system is made up of n workers, labeled 1, 2, $\cdots$, n, forming the set of **N**. Based on the characteristics of inter-cloud WAN connections, CREW abstracts the inter-worker WAN as a fully connected graph G, in which, the directed connection from worker i to worker j is with the available bandwidth of $b_{i,j}$ for *partial reduce*; because of the presence of coexisting traffic, $b_{i,j}$ is probably unequal to $b_{j,i}$. For ease of description, given a worker i, we let its $b_{i,i}$ be a large value thus it would not be the bottleneck. To make full use of the processing capacity of all workers and all the available bandwidth, when each worker completes its local training, saying i for instance, it first serializes its model parameters W_i into numbered chunks with a fixed size, and groups them into n non-overlapping blocks (i.e., $W_i^1, \cdots, W_i^j, \cdots, W_i^n$) for synchronization, respecting the model splitting schemes given by the controller. Then, the j-th block W_i^j will be scattered to worker j. Note that, the scatter of W_i^i will not trigger network transmissions since the destination and the source are the same. On getting these blocks, the receiver caches them first and conducts the reduce computation when getting notifications from the controller.

Following these designs, for each *partial reduce* task, workers in CREW have two types of roles, namely ***initiator*** and ***aggregator***, respectively. Here, *initiator* means that this worker is one of the p workers that have initiated/triggered this round of *partial reduce*; and *aggregator* means that this worker contributes to the reduce computation of this task. Let **P** denote the set of p workers that initiate this round of *partial reduce*. Obviously, by default, for a *partial reduce*, all workers in its **P** would work as both *initiator*s and *aggregator*s at the same time, while all other workers would only work as *aggregators* (Exceptions occur in the case of network outages and refer to §IV-B for details). When a worker gets the configuration of **P** from the controller, it would aggregate the specified p blocks whose sender IDs are exactly corresponding to **P** (Figure 2b), or later in case that some blocks are not obtained yet. Then, the aggregation result would be multicasted back to the p workers in **P** once the computation is done (Figure 2c).

To release the full power of the above designs, however, the following problems must be addressed carefully.

1) How to split the model into blocks to maximize the usage of the available bandwidth?
2) How to deal with possible network outages?
3) How to schedule concurrent transfers for accelerating the completion of synchronization?

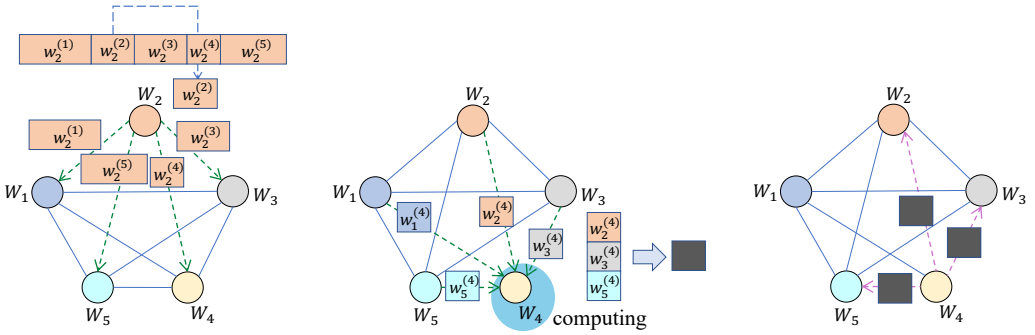
CREW overcomes them with novel algorithms and the design details follow in the next section. Despite that CREW relies on a logical controller by design, the job of the controller is minimal, i.e., 1) computing a default scatter plan based on the network status for all workers, 2) dynamically calculating the optimized *partial reduce* education plan for ready workers once their number reaches p, and 3) dynamically picking replacement workers for failover. In the process, the controller only needs to exchange small-sized control and heartbeat messages with workers. Accordingly, the controller is less likely to become the performance bottleneck. Even if the control messages are delivered with delays, as we will explain in Section IV-A, CREW supports non-blocking scatter designs, thus the impacts of such delays are also non-trivial. Moreover, if the number of workers is huge, a promising solution is to execute the synchronization hierarchically rather than conducting flat model synchronizations. How to use CREW for such scenarios is an interesting future direction.

IV. CREW Algorithms

Now, we explain the algorithm details of CREW.

A. Bandwidth-aware Block Generation

1) Workload splitting for ready workers: In CREW, each worker has direct (logical) WAN connections to all others and it employs all workers to execute the operations involved by each *partial reduce*. Consider the case that workers need to perform *partial reduce* for a group of workers **P**. Intuitively, each worker in **P** should split its model parameters in the proportion of its up- and down- link bandwidths to others for the best usage of the corresponding WAN connections' capacities. However, if each worker only considers the bandwidth of connections it involves, inconsistent scatters of the model would



(a) weighted scatter (b) controlled reduce computation (c) way-back multicast

Fig. 2: The 3-phase workflow of CREW.

occur, which means that the same parameter would be placed into different blocks by different workers. Thus, CREW has to provide a splitting scheme in a network-aware way and ensure that all workers in $\mathbf{P}$ adopt the consistent scheme. To address this, CREW computes the model's splitting plan upon a global view of the involved connections' available bandwidth at the controller. Without loss of generality, consider that the model is with the size of 1 unit, and let x_j be the size of the j-th partition $W_i^j (1 \le j \le n)$ for the scatter. Then, we have $\sum_{i=1}^{n} x_i = 1$; and the vector of $\boldsymbol{x} := (x_1, x_2, \cdots, x_n)$ denotes the weights used by scatter. For bulk transfers triggered by large-size models, their completion times are dominated by their volumes and available bandwidth. Accordingly, the time costs of the weighted scatter and way-back multicast phases specified in Figure 2, can be formulated by Equations (1a) and (1b), respectively. Here, $T^i_{scatter} := \max_{j \in \mathbf{N}} \frac{x_j}{b_{i,j}}$ and $T^j_{mutlicast} := \frac{x_j}{\min_{i \in \mathbf{P}} b_{j,i}}$ denote the minimum time that it takes for the worker i to complete its scatter and for the worker j to complete its multicast, respectively.

$$T_{scatter} := \max_{i \in \mathbf{P}} T^i_{scatter} := \max_{i \in \mathbf{P}} \max_{j \in \mathbf{N}} \frac{x_j}{b_{i,j}} = \max_{j \in \mathbf{N}} \frac{x_j}{\min_{i \in \mathbf{P}} b_{i,j}} \tag{1a}$$

$$T_{multicast} := \max_{j \in \mathbf{N}} T^j_{multicast} := \max_{j \in \mathbf{N}} \frac{x_j}{\min_{i \in \mathbf{P}} b_{j,i}} \tag{1b}$$

Let T_{CREW} be the communication time under the schedule of CREW and define $T = T_{scatter} + T_{multicast}$. Then, we have $T_{\text{CREW}} \ge T_{scatter}$ and $T_{\text{CREW}} \ge T_{multicast}$, which further yield $T_{\text{CREW}} \ge \frac{1}{2}(T_{scatter} + T_{multicast}) = \frac{T}{2}$. When the trained model is large, by splitting the tensors into fine-grained chunks for pipelined communication, the multicast procedure can be partially overlapped with the scatter, thus $T_{\text{CREW}} \le T$. Putting the above analysis together, we obtain $\frac{T}{2} \le T_{\text{CREW}} \le T$. Note that, despite $T_{\text{CREW}} \le T$, in cases like $b_{i,j} = b_{j,i}$ for all worker pairs, the optimal vector $\boldsymbol{x}$ for T also leads to the optimal T_{CREW}. Motivated by this, CREW solves the *linear programming* (LP) of (2) to obtain the optimal weight vector $\boldsymbol{x}$, and uses it as a guideline for the optimization of T_{CREW}.

Algorithm 1 Split Model from Scratch

Require: scatter weights: $(x_1, \cdots, x_n)$; chunk number: k
Ensure: model splitting scheme: $(C_1, \cdots, C_n)$

```
y ← 0; l ← 0
for i ← 1, ··· , n do
    y ← y + x_i
    u ← y * k
    C_i ← {j ∈ Z : l < j ≤ u}  ▷ chunk indexes in block i
    l ← u
end for
```

$$\textbf{Minimize} \quad T_{scatter} + T_{multicast}$$
$$\text{s.t.} \quad \sum_{i=1}^{n} x_i = 1, \text{where } x_i \ge 0 \tag{2}$$

Suppose that the model parameter has been serialized into k chunks, indexed by $1, 2, \cdots, k$, respectively, where $k \gg n$; let C_i $(i = 1, \cdots, n)$ be the sets of chunk indexes belonging to the i-th block; and let the vector of $\boldsymbol{C} := (C_1, \cdots, C_n)$ denote the related model splitting scheme. Then, the produce of how the CREW controller generates a fresh model splitting scheme to approximate the weights given by $\boldsymbol{x} := (x_1, \cdots, x_n)$ is as Algorithm 1 shows. As $\sum_{i=1}^{n} x_i = 1$, all chunks will finally find their blocks. Regarding the value of $\boldsymbol{x}$, it is easy to obtain for a given *partial reduce* task by solving the LP with commercial-off-the-shelf solvers like Gurobi and Mosek.

2) From blocking to non-blocking: According to (2), to obtain the best execution plan (i.e., scatter plan) for a *partial reduce* task, the CREW controller has to know $\mathbf{P}$, the exact set of workers that participate in this round. In practice, workers are not likely to complete a round of training at the same time because of the phenomenon of stragglers [4, 7] For this limit, the straightforward solution is to let each worker start the scatter transmission only after it obtains the exact *partial reduce* plan from the controller; before that, these WAN connections can not be used for this task. We call such a design *blocking scatter*. Recall that, in CREW, each *partial reduce* would employ all available workers and their connections to execute the involved operations; hence, the receivers of each transfer involved in a *partial reduce* task are determined, even when the $\mathbf{P}$ is not generated yet. Accordingly, once a

Algorithm 2 Adjust Model Splitting (i.e., Rearrangement)

Require: new weights: $(x_1, \cdots, x_n)$; current $(C_1, \cdots, C_n)$
Ensure: updated model splitting scheme: $(C_1, \cdots, C_n)$
1: $y \leftarrow 0$; $l \leftarrow 0$
2: $L \leftarrow [\,]$ ▷ record chunk indexes that should be moved
3: **for** $i \leftarrow 0, \cdots, n$ **do**
4: $y \leftarrow y + x_i$
5: $u \leftarrow y * k$
6: $d_i \leftarrow |C_i| - (u - l)$ ▷ number of chunks to move out
7: $l \leftarrow u$
8: **if** $d_i > 0$ **then** ▷ should move out d_i chunks
9: Move the d_i largest indexes from C_i to L
10: **end if**
11: **end for**
12: **for** $i \leftarrow 0, \cdots, n$ **do**
13: **if** $d_i < 0$ **then** ▷ should move in $-d_i$ chunks
14: Move the $-d_i$ smallest indexes from L to C_i
15: **end if**
16: **end for**

worker is ready, it can start its scatter transmission as both the involved data and endpoints are available. We call this a *non-blocking scatter* design, which yields possible improvements upon the *blocking scatter*, by making more efficient use of all the available bandwidth of WAN connections.

As the best scatter weights for a *partial reduce* task could be determined only after all the involved workers belonging to the same *partial reduce* group are determined, CREW uses the design of pre-configuring workers with a default scatter plan, such that workers can start non-blocking scatter transmissions before the best scatter plan is available. In CREW, the controller computes the default values of $\boldsymbol{x}$ and $\boldsymbol{C}$ by assuming $\mathbf{P} = \mathbf{N}$. When a worker is ready, it immediately starts the scatter following the default $\boldsymbol{C}$ and reports its readiness to the CREW controller; once the controller determines the $\mathbf{P}$ for this *partial reduce* task, it first computes the best scatter weight $\boldsymbol{x}$ from $\mathbf{P}$, then adjusts the current model splitting scheme $\boldsymbol{C}$ to generate the new splitting via Algorithm 2, and finally notifies the $\mathbf{P}$ along with the new $\boldsymbol{C}$ to all involved workers. On obtaining $\mathbf{P}$ together with the new $\boldsymbol{C}$, workers adjust their scatter processes correspondingly: if a worker has already received some chunks that are no longer belonging to it, the worker will discard them. To reduce the number of chunks that would get discarded, during the scatter, workers send chunks in increasing order of their indexes; and when the controller has to move some chunks out of a block, as Line 9 in Algorithm 2 shows, it prefers to move chunks in decreasing order of their indexes as chunks with the higher indexes are not transmitted yet.

3) Time complexity: Regarding the time complexity, both the above algorithms are very efficient. In Algorithm 1, the chunk indexes allocated to a block are exactly successive thus constituting a segment. Accordingly, the worst-case time complexity of Algorithm 1 is $O(n)$, since we only need to maintain the lower and upper bounds for each block. Here n is the number of available workers. As for Algorithm 2, the key is to 1) move each block's excessive upper segment of chunk indexes (if any) to L, then 2) empty L by moving these involved chunk indexes back to proper blocks. It is obvious that each chunk would be moved at most once during such a process. Accordingly, the worst-case time complexity of Algorithm 2 is $O(k)$, where k is the total number of chunks. Indeed, by also representing the moved segments of chunk indexes using ranges, it is possible to improve further the time complexity of Algorithm 2 to approximate $O(n)$. To drive Algorithms 1 and/or 2, CREW also needs to compute the scatter weights $\boldsymbol{x}$ by solving LP (2). Currently, commercial-off-the-shelf solvers like Gurobi are efficient in doing this in polynomial time. As CREW supports non-blocking scatter, such a time cost can be masked by the communication.

B. Failover Schemes

During the training, a CREW worker might lose its connection(s) to the controller or other workers, because of the outage of the network or servers. Such link and node outages can be detected by the involved participating servers using schemes like heartbeat and timeout [39]. The failure of a server node results in the breakdown of all the involved active connections. We assume that both endpoints of a connection obtain a consistent view of the possible outage, and present the design principles that CREW could use for these issues.

On the worker side, if a worker finds itself cannot talk to the controller anymore, it then switches to the standalone training mode without triggering new model synchronizations and tries to recover the connections proactively. In the standalone mode, the worker tries to complete the *partial reduce* tasks that it already gets the corresponding $\mathbf{P}$s from the controller; but for other incoming scatter transmissions, it would notify the senders to reject, as it could not guarantee consistent *partial reduce* operations due to the lack of $\mathbf{P}$. For the aforementioned ongoing *partial reduce*, if the worker finds its connections to some other workers are broken, it would not try to fix the impacted task(s) since the controller is unavailable. Once its connection to the controller is recovered, the worker switches back to normal mode.

When a worker (e.g., i) finds that its connection to another worker (e.g., j) is (still) lost, it tries to recover, and inquires the controller for a fixing plan after obtaining a $\mathbf{P}$. If its connection to the controller is lost as well and it happens to be an *initiator* of a running *partial reduce*, it would directly abandon this task; and to accelerate the process of failover, it would also notify all other workers to abandon this task via active connections. If the connection to the controller is fine and this worker i does be an initiator for a running *partial reduce* task (i.e., it belongs to a $\mathbf{P}$), on getting its report, the controller would find a new worker that has live connections to all workers in the same $\mathbf{P}$ to take over the aggregator role of worker j. Let S be the set of candidate workers that have available connections to all workers in $\mathbf{P}$, denote the total block volume that worker i is scheduled to reduce by l_i, and suppose that the j-th block is with the size of $\hat{x}_j$. Then, the CREW controller would pick the one with the minimum estimated workload computed by Equation (3), as the replacement. In case the controller cannot find a replacement, it notifies all the involved workers to discard this round of *partial reduce*.

$$h(S) := \arg\min_{k \in S} \max_{i \in \mathbf{P}} \frac{l_i + \hat{x}_j}{\min(b_{i,k}, b_{k,i})} \quad (3)$$

On the controller side, when a worker is lost, the controller would remove it from the queue that records these ready-yet-unsynchronized workers if it is in, ensuring that this worker would not be grouped into any incoming *partial reduce* task (e.g., **P**). Besides, the controller would reconstruct the abstracted network topology G, and notify all active workers of the results, such that, workers would avoid using this "offline" worker in their incoming scatters. On getting a report of a broken connection, saying that worker i reports that its connection to worker j is broken, as described above, if the reporter worker i is an *initiator*, the controller will reply with either the replacement of j it has found or an abandon message. Note that, in case the controller has received multiple reports of broken inter-worker connections, it would react in the First-Come-First-Serve (FCFS) mode.

Regarding the failover of the CREW controller, since it can be implemented in a stateless manner, once going down, the backup CREW controller will take over the job gracefully by re-obtaining the needed information from workers again. During this handover, ready workers that have already received their *partial reduce* execution plan(s) (e.g., **P**), would not get impacted; and for those that have not determined the execution plan(s), thanks to the non-blocking scatter design of CREW, workers could conduct the scatter following the default scheme until they obtain the optimized plan(s) from the new controller. If worker failure also occurs at the same time, the remaining workers could discard this round of *partial reduce* if they have not received the failover scheme from the new controller within a given time. Indeed, to make CREW self-healing and able to recover from all possible uncovered corner cases, workers would discard a running *partial reduce* task if it has not been completed within a given time budget.

C. Flow Scheduling Principles

In CREW, multiple flows belonging to different *partial reduce* tasks might use the same inter-cloud connections at the same link, resulting in bandwidth competition. For example, when a worker completes a task's reduce computation, it would multicast back the result, occupying inter-cloud connections. Very soon, this worker might complete its local training in a short time, split its trained model into blocks, and send one in that connection, too. Even worse, the execution of the link failover scheme might make the case more complicated. According to the per-flow max-min fairness bandwidth allocation that the network provides by default, these two transfers would share the inter-cloud link capacities equally. This is not conducive to minimizing the average completion time of the entire *partial reduce* tasks.

For such a scheduling problem, as concurrent *partial reduce* tasks involve the same size of the model to synchronize and a *partial reduce* task is started only when all their members are ready, just like the schedule of coflow [40], a straightforward design is to prioritize their transfers in an FCFS (First Come First Serve) mode at the level of the task. However, we argue that scheduling transfers with FCFS at the level of flow is better for CREW. Consider that, two flows f_i and f_j belonging to the i-th and j-th *partial reduce* tasks, respectively, are going through the same link/path from worker s to worker d. Let us assume that, the controller issues the j-th task before the i-th one (i.e., $i > j$), while at the source sender f_i is ready to start before f_j. Note that, in CREW, only *initiator* workers could scatter and a worker can work as the *initiator* for only one task at the same time, while multicasts are triggered by aggregators and a worker can be the aggregator of multiple tasks at the same time. Accordingly, there are only three possible cases for the aforementioned concurrency examples: f_i belongs to scatter while f_j belongs to multicast (**C1**); f_j belongs to scatter while f_i belongs to multicast (**C2**); or both are multicast flows (**C3**). To prioritize flows respecting the order of their *partial reduce* tasks, the transmission of f_i would be preempted, by f_j. This would 1) let the worker s act as a straggler for task i (in the view of worker d) in the case of **C1**, or 2) prevent worker d from moving to the next round of training because the multicast is blocked in the case of **C2**. As for the case of **C3**, f_i and f_j are equally important to worker d, only after receiving both f_i and f_j can worker d enter into its next round of training, no matter which type of scheduling CREW uses. Motivated by these facts, workers in CREW schedule concurrent transfers with FCFS locally (i.e., flow-level). As Section V will show, we verify and confirm the benefits of such a design with simulations.

The above FCFS flow scheduling principle also makes the performance of CREW insensitive to the delays introduced by failover schemes. Specifically, the chunks of the scatter operation for failover would only get sent on a connection after all other scheduled chunks have been delivered. As a result, before a link/node failure is detected and the failover plan takes effect, active workers execute their planned transmission tasks normally, without wasting bandwidth.

Modern cloud data centers generally provide guaranteed performance, e.g., with service level agreement (SLA), for deployed applications by design. Thus, the overload and effect on and from the workflows of other applications running on the same clouds (i.e., inter-application interference) can generally be controlled to some extent, if unavoidable. Even for the possible congestion on the inter-cloud connections, CREW relies on existing transport protocols like TCP for the underlying transmission of each flow [41]. These protocols are equipped with advanced congestion control designs, enabling CREW to react to possible network congestion and make usage of available remaining bandwidth adaptive. Recent studies have reported that inter-cloud connections generally have very stable capacities [41]. Even if the actually obtained bandwidth does not match with what it is believed to have, as the result of Section V-H has shown, CREW could still work properly.

V. Performance Evaluation

In this section, we evaluate the performance of CREW under four different workloads, constructed by using real-world WAN bandwidth distributions [42] and model training time distributions [7]. Results imply that,

1) By leveraging the entire network's bandwidth, CREW obtains up to dozens of times of performance improvements compared to PS, Ring, TOPOADOPT, and BLINK for the execution of a single *partial reduce* task, and up to $10-15\times$ and $4-5\times$ speedups compared to the all-to-all direct delivery scheme and original *partial reduce* design, respectively, for the iterative training of models.
2) According to the experimental results of link bandwidth allocation, we demonstrate that the flow scheduling design of FCFS is simple-yet-efficient.
3) Meanwhile, CREW can cope well with network outages; and due to its load balance design, CREW effectively alleviates performance degradation caused by increasing communication traffic.
4) Moreover, we also conduct experiments in computation-intensive and communication-intensive scenarios, the results indicate that CREW still performs well.

A. Methodology

Baselines and evaluation tools. In tests, we mainly use the following schemes that could achieve consistent *partial reduce* operations for a given group of workers **P**, as the baselines to study the performance of CREW:

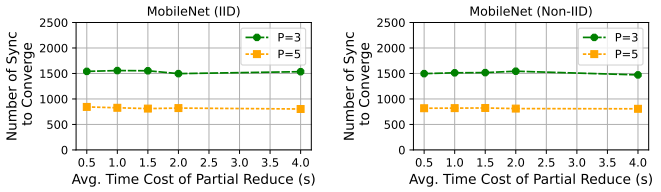
- PS [17]: the worker having the largest (direct) average bandwidth to all other workers is selected as the global PS; if there are multiple candidates, the one with the smallest average latency to others is preferred.
- Ring [31]: a ring is generated for **P** greedily—starting from a random worker in **P**, then continuing to pick the new worker in **P**, which has the largest bandwidth to the last processed worker, until a ring is found.
- BLINK: following [43], multiple spanning trees are generated for **P**, and the workload of *partial reduce* is distributed among five of them, by solving a mixed integer linear programming (MILP) problem;
- ToA (TOPOADOPT): the spanning tree generation and workload distribution algorithm, based on simulated annealing and proposed by [44], is extended to support the *all reduce* requirement of **P**, straightforwardly.
- *A2A (All-to-All direct delivery)*: each worker in **P** directly sends its model parameters to other $p-1$ workers;
- *OPR (Original Partial Reduce)*: workers in **P** split the model parameters into p blocks and each worker in **P** is responsible for aggregating one of the blocks. Indeed, such a design can be treated as a distributed version of the original parameter server based model synchronization [17], in which each worker acts as a sharded parameter server at the same time.

The difference between OPR and CREW is that the former only employs p workers while the latter splits the model parameters into n blocks and employs n workers. To evaluate these schemes, we have developed a flow-level simulator with Python 3, which would precisely simulate the aforementioned synchronization under various execution schemes. We employ Python-MIP to obtain the linear programming results of Equation (2), which are used for the network-aware model splitting (i.e., *Bandwidth-aware Block Generation*) for both CREW and OPR. By default, CREW conducts blocking scatter when executing partial reduce tasks; we also investigate the benefits of non-blocking scatter design in detail. Since today's network design adopts per-flow max-min fairness (i.e., FS) for the allocation of bandwidth, we first compare the performance of the three schemes under FS and flow-level FCFS bandwidth allocation in a case study, and then give a further analysis of FCFS in the subsequent experiments.[1] Finally, the impact of inaccurate bandwidth estimation is also studied.

Network and workloads. We consider a Geo-DML environment involving n workers. Extensive tests indicate that CREW could consistently outperform baselines under different system scales. In the performance study of related works [2, 6, 9, 45, 46], the scale of Geo-DML tasks generally ranges from several to tens, and to over a hundred. As case studies, following them, we use 60, roughly their average value, as the default value of n, to show results in this section. To simulate the real network environment, the available bandwidths of inter-cloud connections (i.e., $b_{i,j}$) are sampled from the inter-datacenter bandwidth measured from Amazon EC2 [42]. Meanwhile, we also set up another bandwidth distribution for connections between workers, which are calculated by $k * u$, where k is an integer in the range of $[1, 10]$ sampled from $N(5, 1)$ and u is the minimum bandwidth unit with the value of 25Mbps. The former network setting is labeled with N_1, and the latter is labeled with N_2. Note that $b_{i,j}$ and $b_{j,i}$ between worker i and worker j may not be equal. Regarding the model's local training computation time, we consider both the time distributions of training an LSTM model on UCF101 (denoted by T_1), and the time distributions of training a Transformer model on WMT16 (denoted by T_2), reported by [7]. To highlight the comparison, we re-scale both distributions' sample computation time values to the $[0.05, 0.2]$s range. In some instances, we also increase the computation times by $2-6\times$ to study the performance of CREW under different training workloads. Here, the variation of training time is used to simulate *straggler* workers in heterogeneous environments. As we have two network settings N_1 and N_2, by using different combinations of N_1, N_2, and T_1 and T_2, we consider 4 workloads in tests: (N_1, T_1), (N_1, T_2), (N_2, T_1), and (N_2, T_2). Regarding the model size (which is related to the traffic load in model synchronization), according to [47], we consider the number of parameters of the model at the millions level. So, we assume the model has a default size of 180MB. In communication-intensive scenarios, we would increase the model size by $2\times$, $4\times$, and $8\times$.

Metrics. In practice, the efficiency of distributed training is jointly determined by two factors: 1) the number of training rounds needed to meet the stopping conditions like reaching a target accuracy, or a predefined number of training rounds, and 2) the average time of conducting a round of training. Generally, in the context of *partial reduce* enabled heteroge-

[1] Note that, even for OPR, because of the asymmetric bandwidth of an inter-worker connection, FCFS would take effects: the new broadcast transfer from a worker might compete for a connection with slow ongoing scatter transfer.



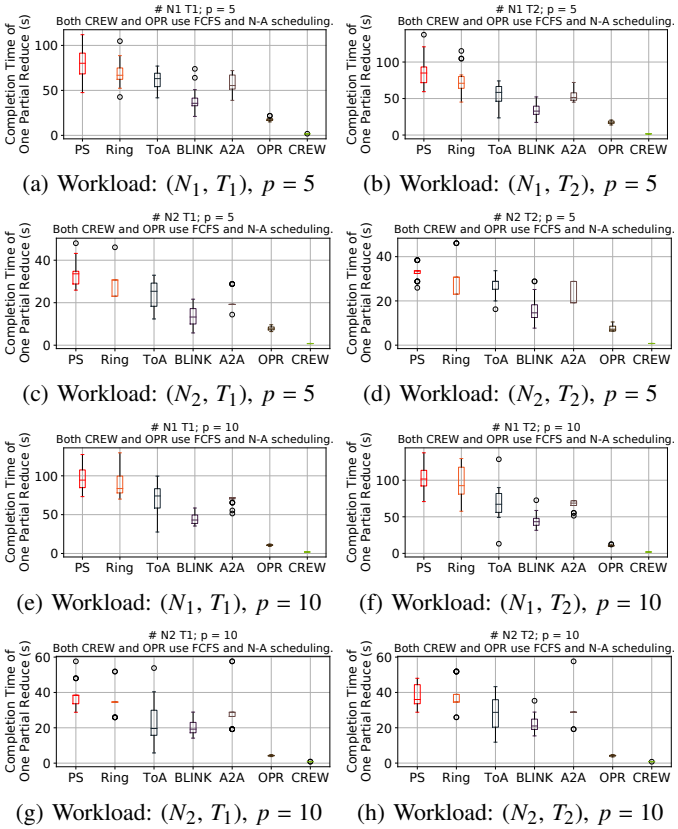
(a) MobileNet (IID) (b) MobileNet (Non-IID)

Fig. 3: When 8 workers jointly train MobileNet using partial reduce, no matter whether the distribution of the MNIST dataset is IID or non-IID, all workers take similar rounds of synchronization to reach the targeted accuracy of 0.95.

neous DML, workers dynamically form *partial reduce* groups in a random manner. Reducing the time cost of each round of *partial reduce* model synchronization under the same consistent hyper-parameter settings (including using the same P value) [4, 16], would not change the statistical characterization of how workers form *partial reduce* groups, and thus does not touch the convergence behaviors of the training (e.g., the number of training rounds to reach the target accuracy). As an empirical study, we have employed our resource-limited server to emulate the case that 8 workers use *partial reduce* based model synchronization to train the model of MobileNet [48] over the dataset of MNIST. To highlight the impacts of the efficiency of *partial reduce* on the convergence behavior of workers, we artificially set the completion time of each *partial reduce* task to t seconds, which is randomly selected from the range of $[0.8x, 1.2x]$ with equal probability. Here, x denotes the average time cost of *partial reduce* operations and ranges from 0.5 to 4.0 in tests. Regarding the distributions of the training samples, besides Independent and Identically Distributed (IID), we also test the scenario that the training dataset on workers is non-IID to emulate the case of federated learning [49]. To reduce the impacts of noises introduced by the randomness of the distributed training, for the same x and p, we re-run the training 10 times to calculate their average values. As the results in Figure 3 show, for both IID and non-IID data, all workers require consistent rounds of *partial reduce* operations to reach the same target model accuracy of 0.95. Given that CREW aims at accelerating the execution of *partial reduce* tasks rather than reducing the number of training rounds to converge, we evaluate its performance using direct metrics from the perspective of communication. Thus, for all comparative schemes in tests, besides the time cost of executing a round of partial reduce, we also assess them by the number of training rounds each worker could complete in a fixed period of time (i.e., with the default value of 50s in our tests)—a large value means a faster synchronization. For each parameter setting, we conduct at least 10 trials and describe their distributions or compute the average values.

B. Completion of Partial Reduce

As Figure 4 shows, across all workloads, compared with all the baselines, CREW achieves the best performance in terms of the time of completing a *partial reduce* task. Indeed,



(a) Workload: (N_1, T_1), $p = 5$ (b) Workload: (N_1, T_2), $p = 5$

(c) Workload: (N_2, T_1), $p = 5$ (d) Workload: (N_2, T_2), $p = 5$

(e) Workload: (N_1, T_1), $p = 10$ (f) Workload: (N_1, T_2), $p = 10$

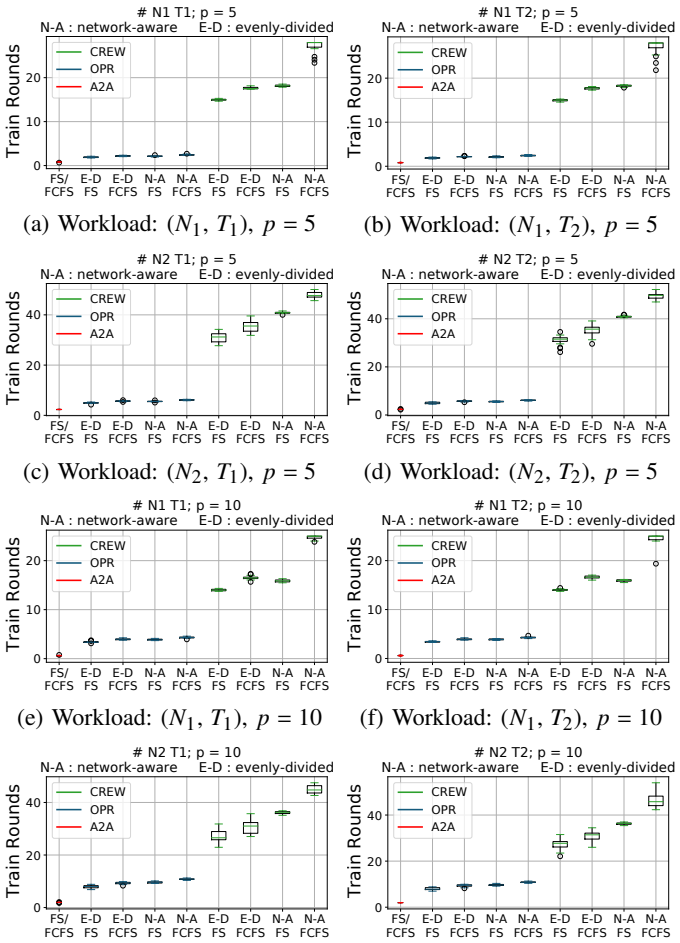
(g) Workload: (N_2, T_1), $p = 10$ (h) Workload: (N_2, T_2), $p = 10$

Fig. 4: Compared with PS, Ring, ToA, BLINK, and A2A, both OPR and CREW need much less time (up to dozens) to complete a round of *partial reduce* operation. Moreover, beyond OPR, CREW accelerates the execution further, by making use of all active workers along with their connections.

even OPR, i.e., the variant of CREW that only distributes the workload of a *partial reduce* task to its participants, can greatly outperform other baselines including BLINK and TopoAdopt. Here, both OPR and CREW schedule concurrent flows following the FCFS policy and distribute the workloads among workers in a network-aware (N-A) manner. Such results imply that, despite solutions built upon spanning tree packing (e.g., BLINK [43] and TopoAdopt [44]) can make more efficient usage of the abundant network connections than solutions relying on a single PS or Ring in theory, their performances highly depend on the abilities of the proposed algorithms—the results of BLINK and TopoAdopt are far from optimal and there is a large room for improvement. Actually, due to their complex algorithm designs, both BLINK and TopoAdopt are time costly. In contrast, CREW is fast and achieves the best completion time—it is specialized to conduct optimized *partial reduce* operations over fully-connected workers and the most complicated part is to solve a simple LP.

C. Training Iterations

Next, we investigate the number of training iterations workers would obtain for a period of 50s, under various *partial reduce* implementation schemes. As the above results have shown, OPR outperforms PS, Ring, BLINK, and TopoAd-



(a) Workload: (N_1, T_1), $p = 5$ (b) Workload: (N_1, T_2), $p = 5$

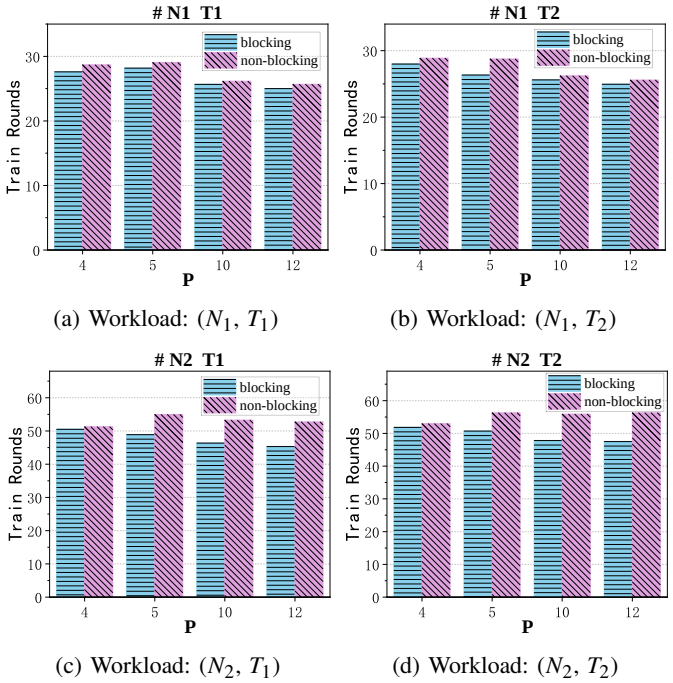
(c) Workload: (N_2, T_1), $p = 5$ (d) Workload: (N_2, T_2), $p = 5$

(e) Workload: (N_1, T_1), $p = 10$ (f) Workload: (N_1, T_2), $p = 10$

(g) Workload: (N_2, T_1), $p = 10$ (h) Workload: (N_2, T_2), $p = 10$

Fig. 5: CREW *vs* A2A *vs* OPR under 4 workloads when $p = 5$ and 10, respectively. Note that, CREW uses n workers for each *partial reduce* while OPR uses p workers. Results confirm that, the novel designs of CREW enable it to outperform all the other two baseline schemes significantly.

OPT; thus, we use A2A and OPR as the baselines here. To compare with network-aware model splitting and FCFS flow scheduling, we also modify OPR and CREW to just evenly split the model parameters into p and n blocks, and schedule concurrent flows with FS. As a result, for each of them, we have 4 variants: (E-D, FS), (N-A, FS), (E-D, FCFS), (N-A, FS). But for A2A, there is no difference between FS and FCFS scheduling since it would not encounter flow contentions. As shown in Figure 5, where p is set to 5 and 10 for the first four figures and the last four figures, respectively, CREW can always significantly speed up the model synchronization process regardless of workload, and complete more model training rounds during the same time. Compared with A2A and OPR, when p is 5, CREW achieves up to $12-15\times$ and $8-9\times$ speedup, respectively. And when p is 10, CREW can be at most $12-15\times$ and $4-5\times$ faster than the other two communication schemes. This is mainly due to that CREW divides the model into n parts and leverages the entire network to help the transmission of model parameters during model synchronization. We can see that the gap between CREW and



(a) Workload: (N_1, T_1) (b) Workload: (N_1, T_2)

(c) Workload: (N_2, T_1) (d) Workload: (N_2, T_2)

Fig. 6: Compared with *blocking scatter*, the design of *non-blocking scatter* always brings performance improvements (up to $1.2\times$ speedup) to CREW in tests.

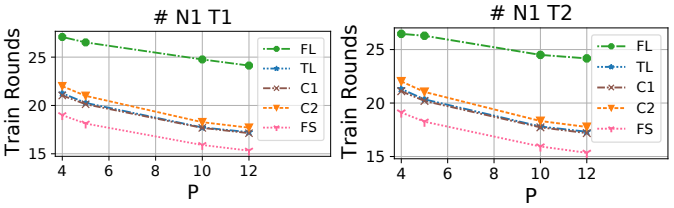
OPR is getting smaller as p goes up. This is because, with the increase of p, OPR can split the model parameters into more blocks and employ more workers for model synchronization. Once the value of p increases to n, the results of OPR would be completely equivalent to that of CREW. In CREW, network-aware model splitting and flow-level FCFS design can always efficiently increase the train rounds compared to evenly-divided and FS. When both apply network-aware model splitting and flow-level FCFS in CREW, it is about $1.5-2\times$ faster than default evenly-divided and FS.

D. Benefits of Non-blocking Scatter

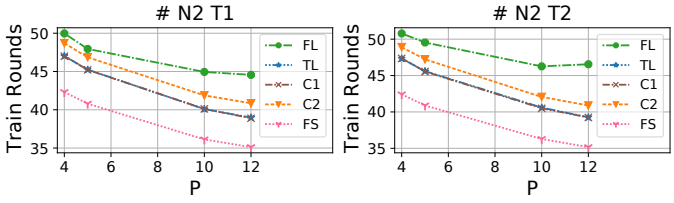
As mentioned in §IV-A2, the *non-blocking scatter* design can further improve the bandwidth utilization of CREW. Now, we verify this with experiments. As shown in Figure 6, regardless of the workload, the *non-blocking scatter* design always brings some performance improvements, completing more training rounds in a fixed period of time, and achieving up to $1.2\times$ speedup compared to *blocking scatter*. However, in a few scenarios, the performance gain is slight, which is related to the network configuration, straggler distribution, and the value of p. Since the performance of the *non-blocking scatter* design is highly dependent on the workload and network environments, by default, CREW uses the *blocking scatter* implementation in other experiments.

E. Impact of Flow Scheduling Scheme

In §IV-C, we have discussed the possible scenarios of concurrent flows in a link, and we indicate that because of the importance of each task self, the flow that occurs

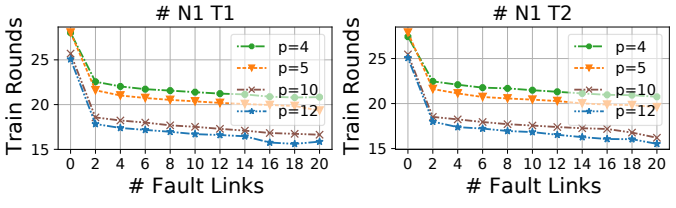


(a) Workload: (N_1, T_1) (b) Workload: (N_1, T_2)

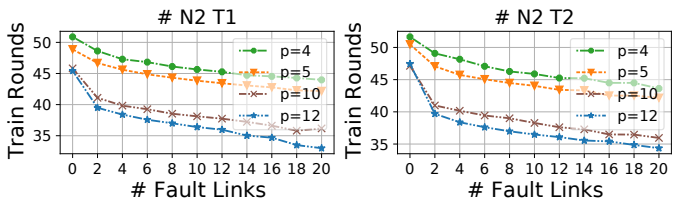


(c) Workload: (N_2, T_1) (d) Workload: (N_2, T_2)

Fig. 7: The performance of CREW when different flow scheduling schemes are used and the network-aware model splitting design is enabled. Results confirm that, for the flow scheduling problem CREW involves, flow-level FCFS (**FL**) is good enough, outperforming **FS**, task-level FCFS (**TL**), and the traffic type based prioritized scheme of **C1** and **C2**.



(a) Workload: (N_1, T_1) (b) Workload: (N_1, T_2)



(c) Workload: (N_2, T_1) (d) Workload: (N_2, T_2)

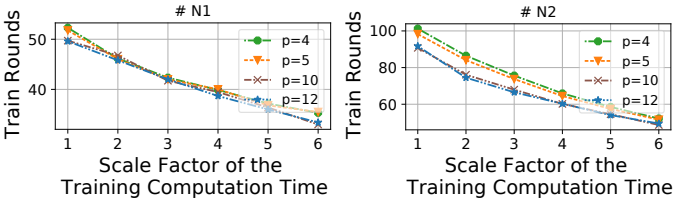
Fig. 8: The performance of CREW under different degrees of network faults while the network-aware model splitting design is always enabled. Results show that CREW is able to deal with possible network outages properly and CREW's scheme of bandwidth-aware load balancing scheme can largely reduce the impacts of network outages.

first in a link should not be preempted by the flow that occurs later, otherwise, it may cause stragglers in **C1** case or prevent receiving workers from turning to the next training round in **C2** case (i.e., task-level FCFS, or **TL** for short). And we argue that the flow-level FCFS principle (or **FL** for short) for bandwidth allocation can achieve the optimization of training QoS. To verify this idea, we carry out experiments to simulate the scenarios of **C1** and **C2**, respectively. In **C1**, we just simulate a flow belonging to multicast preempts a flow belonging to scatter, which is opposite in **C2**. Besides, we also compare the task-level FCFS design where the bandwidth is preferentially allocated to the flow with a smaller *pid* value, where *pid* is the index (with an increasing value) given by the controller, identifying which *partial reduce* task this transfer (flow) belongs to, which means that flows generated by earlier *partial reduce* tasks obtain bandwidth preferentially.

Here, we present the comparison results under 5 flow scheduling schemes and evaluate them by the number of training rounds completed at the same time. As shown in Figure 7, the performance of CREW degrades significantly when bandwidth preemption occurs, especially when a flow belonging to multicast preempts the bandwidth of a flow belonging to scatter (i.e., **C1**). For simplicity, let's express this as a multicast flow preempts a scatter flow. This situation gets so bad because the receiving worker (i.e., destination worker d in §IV-C) must collect all p scatter flows from the workers in **P** and can it operate reduce computation (i.e., aggregation) for a block of model parameters. If one of the scatter flows is delayed to delivery, then the p workers in **P** would get stuck waiting for the aggregation result of the block corresponding to that scatter flow. Instead, in **C2**, when a multicast flow is preempted by a scatter flow, it only prevents the receiving worker from getting the aggregation result, the other $p-1$ workers in **P** can still get the aggregation result on time, having relatively less effect on performance. Regardless, preemption always slows down model synchronization in various degrees. Meanwhile, we observe that the performance of the task-level FCFS and preemption of **C1** is very close. This is because, in **C1**, the multicast flow that preempts the scatter flow usually has a smaller *pid*, making its effect approximately equivalent to task-level FCFS. Besides, according to the comparison between flow-level FCFS and FS, we can infer that concurrent flows in a link should also not share the bandwidth of the link, as this is not conducive to minimizing the average completion time. Comparing the results of flow-level FCFS and task-level FCFS, it is found that flow-level FCFS is more practical. Moreover, in all transfer scheduling scenarios, with the increase of p, the number of training rounds that workers can complete decreases. This is because the larger p is, the effectiveness of *partial reduce* to tolerate stragglers will be weakened, and the training speed would slow down, which is consistent with prior work [4].

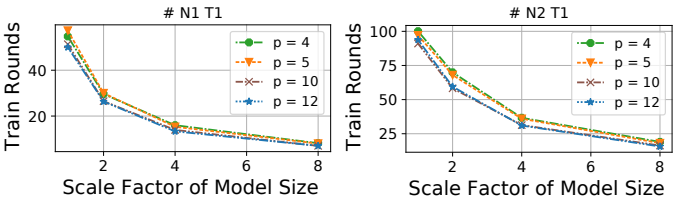
F. Impact of Network Outage

Since connection failures are common in WAN scenarios, we have designed failover schemes to cope with possible network faults in §IV-B. To prove the effectiveness of our scheme, we have evaluated the performance of CREW under different degrees of network faults. By changing the number of faulty links, we can simulate the different degrees of network faults. As shown in Figure 8, at the beginning of the link failure, the performance of CREW decreases significantly, but with the deepening of the network fault degree (more faulty links appear), the performance of CREW decreases very little. This is mainly because we realize the data re-transmission of faulty links based on a load balancing fashion, avoiding using those links that are sending or will send heavy traffic. Even if there are more faulty links, the traffic that needs to be



(a) Network Setting: N_1 (b) Network Setting: N_2

Fig. 9: The performance of CREW under different degrees of training computation times. Results indicate that, for computation-intensive DML, the room for communication optimization is limited and the performance of CREW is related to internal characteristics of the workload.



(a) Workload: (N_1, T_1) (b) Workload: (N_2, T_1)

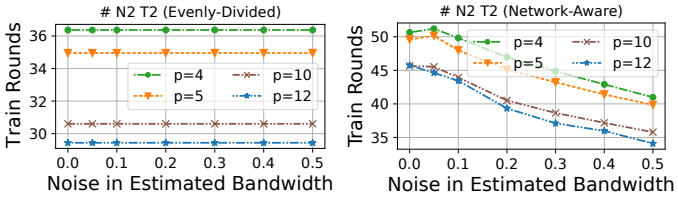
Fig. 10: The performance of CREW under different model sizes. Results imply that, thanks to its novel designs, CREW could reduce the bottleneck impacts of model synchronization for communication-intensive DML significantly.

re-transmitted would be evenly distributed among free links across the entire network, with little performance degradation. We also find that CREW is more susceptible to network faults when p is getting larger, because the larger the p is, the more workers will be involved in model synchronization, increasing the re-transmission traffic.

G. Impact of DML Task Type

To test the performance of CREW under computation-intensive machine learning models, we prolonged the time distribution of T_2 numerically by 2×, 3×, 4×, 5×, and 6×, respectively. In these tests, the training is terminated once the simulation time reaches 100s. As shown in Figure 9, for both $p = 10$ and $p = 12$, the train rounds decrease almost linearly with the increase of training time under the bandwidth distribution of N_1; for both $p = 4$ and $p = 5$, their declines becomes slightly moderate. This is reasonable: with the increase in computation time, the bottleneck of the distributed training shifts from communication to computation; the time cost of training computation would dominate the entire training thus the room for communication optimization that CREW could make use of, is limited in these cases. Compared with the results on N_1, CREW shows a slightly better tolerance on slow down when network distribution follows N_2. And again, different from the results on N_1, the alleviation on N_2 is more obvious with a large p. Such results imply that the speedups that CREW could achieve are also determined by the internal characteristics of the workload.

To test the performance of CREW under communication-intensive machine learning models, we now increase the default model size by 2×, 4×, and 8×, respectively, and test it with four workloads. Like the study of computation-intensive workloads, we also terminate the training once the simulation time reaches 100s. As shown in Figure 10, compared with computation-intensive scenarios where CREW yields little gain, the outperforming of CREW in communication-intensive scenarios is pretty obvious (the results of (N_1, T_2), and (N_2, T_2) are quite similar thus omitted). The number of train rounds does not decrease linearly with the increase in the model size. CREW effectively alleviates performance degradation under all workloads, which verifies that the design of CREW can effectively utilize the bandwidth in the network to accelerate model synchronization.



(a) Workload: (N_2, T_2) (b) Workload: (N_2, T_2)

Fig. 11: With the level of noise in the estimated bandwidth increasing, the completion of *partial reduce* tasks under the schedule of CREW (N-A) is slowed down slowly. In contrast, CREW (E-D) is agnostic to the network bandwidth, yielding a promising solution for unstable network environments.

H. Impact of Inaccurate Bandwidth Estimation

Last but not least, given that in some cases, the available bandwidth of WAN connections might not be perfectly estimated or controlled, we now study the impacts of noise in estimated inter-worker connection capacities on the performance of CREW. To do so, we suppose that the actual capacity of a directed link is b and the estimated value that CREW uses to make decisions is $b(1 + x)$. Here, x is a random variable following the uniform distribution of $[-\lambda, \lambda]$, and the value of λ is increased from 0.1 to 0.5. Despite the completion speed of partially reduce tasks and the impacts of noise partially depending on the type of workload, consistent phenomenons are observed. As the case of workload (N_2, T_2) in Figure 11 shows, the performance of CREW (E-D) is not impacted by the noise, since it distributes the workload to workers agnostic to the network bandwidth; in contrast, with the level of noise increasing, the completion of *partial reduce* tasks under the schedule of CREW (N-A) is slowed down slowly; and as expected, it might be even worse than that of CREW(E-D) for some workloads when the noise is significant (not shown here). Thus, for very unstable network environments, we argue that distributing the workload uniformly is a promising solution.

VI. Related Work

In Section II, we have briefly overviewed the development of Geo-DML and analyzed other *partial reduce* schemes in detail. Now, we have a broader discussion on other straggler-resilient techniques and communication optimization designs.

A. Straggler-Resilient Techniques

Recent works have shown that straggler workers are hard or even impossible to avoid in real-world distributed training, due to the ubiquitous heterogeneities stemming from various aspects like multifarious computation capacities, imbalanced training workloads, and unpredictable resource contention in shared environments [4, 7, 45, 50]. The work of DLion [45] showcases that, for the effects of heterogeneous computation capacities, by dynamically adjusting the batch size settings for workers accordingly, it might be possible to trim their completion times for a round of local training, yielding benefits to data parallelism [45]. However, in some cases, the training dataset samples themselves (e.g., videos, sentences) might have various sizes or lengths, making the training workloads imbalanced and unstable [7]; also, the computation capacities of workers might be unstable because of resource contention (e.g., in shared cloud environments) [7].

To be straggler-resilient, besides *partial reduce*, there are several alternative designs [18, 51]. For example, based on the parameter server framework [17], the work of [52] treats these straggler workers as backup and computes the new global model parameters only based on the results of the first k completed workers, where the value of k can be tuned dynamically. Differently, schemes based on Asynchronous Parallel (ASP) [53] address the issue by removing the barrier between each round of local training. However, as the global model on the parameter server might be aggregated from arbitrarily stale models that introduce non-trivial noises/errors to the training, ASP might make the distributed training non-convergent [51]. To overcome this issue, the scheme of Stale (-aware) Synchronous Parallel has been proposed, which lets fast training workers wait for slow workers, provided the gap between their training rounds exceeds the pre-defined or dynamic-tuned threshold [18, 51]. Similar to SSP, to bound the effects of stale updates in the context of cross-device federated learning, given that there are a large number of heterogeneous and unstable participating workers, instead of blocking the faster workers, solutions like FedSA [54] exclude results that are too stale. Distinguished from the above schemes and following [4], CREW tames the effects of stragglers by only conducting model synchronization for ready workers without relying on centralized parameter servers.

B. Communication Optimization Strategies

Roughly, the principles of alleviating the bottleneck effects of communication for DML can be classified into three orthogonal types: 1) reducing the traffic volume or communication frequency, 2) masking the communication with training computation using parallelization/pipelining, and 3) making efficient usage of all available network capacities.

Currently, gradient quantization, sparsification (e.g., Top-k, random-k), and low-rank decomposition (or sufficient factors) [5, 55] are popular schemes that could significantly reduce traffic volumes. As lossy compression schemes, they generally work with error-feedback (EF) remediation designs. Indeed, some existing *partial reduce* designs like [5, 6, 22] can cut down the traffic load as well. Moreover, from another view, at a high level, by using *partial reduce* or EF-enabled sparsification, the model gradients or parameters are finally synchronized in lower frequencies. Besides these schemes, in-network cache/aggregation is another promising solution to reduce the traffic volume, for both intra-datacenter distributed training and federated learning [20, 56]. As the second type of solution, given that deep neural networks are generally trained layer-by-layer, a genetic design for the optimization of communication is to mask the transmission with computation via pipeline. For example, workers can 1) split their gradient/parameter tensors into partitions, such that parts of the communication can overlap with computation, and at the same time, 2) rearrange the involved transmissions respecting the orders in which these partitions would be used in the forward process so that the overlapping can be further improved [57]. Different from the above schemes, the third type of solution aims at making efficient use of heterogeneous inter-worker connections by 1) executing collective operations (e.g., AllReduce) in a bandwidth- and topology- aware manner (e.g., via multiple spanning trees [43, 44]), or 2) adjusting the inter-worker network topology or/and link capacities respecting their traffic patterns, if supported [58].

At the high level, from the view of the aggregator role of each worker, CREW can be treated as sharing a similar idea of finding multiple spanning trees for AllReduce operations. Nevertheless, compared with BLINK [43] and TopoAdopt [44], CREW is more effective and powerful with an elegant design that supports both non-blocking transmission and fast failover. Besides, enhancing the performance of CREW with other orthogonal techniques like lossy data compression [55] and communication-computation overlapping [22], is promising and interesting. We left them as future directions.

VII. Conclusion

In this paper, we presented CREW, an efficient *partial reduce* implementation design for cross-cloud Geo-DML training. By balancing the involved communication and reduce operations among all available workers respecting the state of inter-cloud connections, and conducting FCFS-alike flow scheduling, CREW could make effective usage of the available inter-cloud network to achieve efficient *partial reduce*.

References

[1] R. Wang, S. Luo, K. Li, and H. Xing, "Efficient partial reduce across clouds," in *Proceedings of the 6th Asia-Pacific Workshop on Networking (APNet)*, Jul 2022, pp. 99–100.

[2] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G. R. Ganger, P. B. Gibbons, and O. Mutlu, "Gaia: Geo-distributed machine learning approaching lan speeds," in *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017, pp. 629–647.

[3] P. Zhou, Q. Lin, D. Loghin, B. C. Ooi, Y. Wu, and H. Yu, "Communication-efficient decentralized machine learning over heterogeneous networks," in *Proceedings*

of the IEEE 37th International Conference on Data Engineering (ICDE). IEEE, 2021, pp. 384–395.

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

[5] P. Xie, J. K. Kim, Q. Ho, Y. Yu, and E. Xing, "Orpheus: Efficient distributed machine learning via system and algorithm co-design," in *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 1–13.

[6] C. Hu, J. Jiang, and Z. Wang, "Decentralized federated learning: A segmented gossip approach," in *Proceedings of the 1st International Workshop on Federated Learning for User Privacy and Data Confidentiality*, 2019. [Online]. Available: http://arxiv.org/abs/1908.07782

[7] S. Li, T. Ben-Nun, S. D. Girolamo, D. Alistarh, and T. Hoefler, "Taming unbalanced training workloads in deep learning with partial collective operations," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2020, pp. 45–61.

[8] L. Liu, H. Yu, G. Sun, L. Luo, Q. Jin, and S. Luo, "Job scheduling for distributed machine learning in optical wan," *Future Generation Computer Systems*, vol. 112, pp. 549–560, 2020.

[9] L. Luo, Y. Zhang, Q. Jin, H. Yu, G. Sun, and S. Luo, "Fast synchronization of model updates for collaborative learning in micro-clouds," in *Proceedings of the IEEE 23rd HPCC; 7th DSS; 19th SmartCity; 7th DependSys*, 2021, pp. 831–836.

[10] P. Zhou, Q. Lin, D. Loghin, B. C. Ooi, Y. Wu, and H. Yu, "Communication-efficient decentralized machine learning over heterogeneous networks," in *Proceedings of the IEEE 37th International Conference on Data Engineering (ICDE)*, 2021, pp. 384–395.

[11] L. Liu, H. Yu, and G. Sun, "Reconfigurable aggregation tree for distributed machine learning in optical wan," in *Proceedings of the 3rd International Conference on Applied Machine Learning (ICAML)*, 2021, pp. 206–210.

[12] M. Potheri, "Multi-cloud machine learning with data from on-premises and training with google cloud vertex platform," https://blogs.vmware.com/apps/2021/11/multicloud_gcp_vertex_part1.html (Accessed date: Jul 13 2022), Nov 2021.

[13] A. Das, T. Castiglia, S. Wang, and S. Patterson, "Cross-silo federated learning for multi-tier networks with vertical and horizontal data partitioning," *ACM Transactions on Intelligent Systems and Technology*, vol. 13, no. 6, pp. 1–27, Sep 2022.

[14] G. Aceto, A. Botta, P. Marchetta, V. Persico, and A. Pescapé, "A comprehensive survey on internet outages," *Journal of Network and Computer Applications*, vol. 113, pp. 36–63, 2018.

[15] V. Giotsas, C. Dietzel, G. Smaragdakis, A. Feldmann, A. Berger, and E. Aben, "Detecting peering infrastructure outages in the wild," in *Proceedings of the ACM SIGCOMM Conference*, 2017, pp. 446–459.

[16] L. Mai, G. Li, M. Wagenländer, K. Fertakis, A.-O. Brabete, and P. Pietzuch, "Kungfu: Making training in distributed machine learning adaptive," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020, pp. 937–954.

[17] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 583–598.

[18] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More effective distributed ml via a stale synchronous parallel parameter server," in *Advances in Neural Information Processing Systems*, C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger, Eds., vol. 26, 2013.

[19] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, E. P. Xing, J. E. Gonzalez, and I. Stoica, "Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning," in *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Jul. 2022, pp. 559–578.

[20] S. Luo, P. Fan, H. Xing, L. Luo, and H. Yu, "Eliminating communication bottlenecks in cross-device federated learning with in-network processing at the edge," in *Proceedings of the IEEE International Conference on Communications (ICC)*, 2022, pp. 4601–4606.

[21] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project adam: Building an efficient and scalable deep learning training system," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 571–582.

[22] P. Watcharapichat, V. L. Morales, R. C. Fernandez, and P. Pietzuch, "Ako: Decentralised deep learning with partial gradient exchange," in *Proceedings of the 7th ACM Symposium on Cloud Computing*, 2016, pp. 84–97.

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

[24] H. Li, A. Kadav, E. Kruus, and C. Ungureanu, "Malt: Distributed data-parallelism for existing ml applications," in *Proceedings of the 10th European Conference on Computer Systems (EuroSys)*, 2015, pp. 1–16.

[25] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, and S. Chintala, "Pytorch distributed: Experiences on accelerating data parallel training," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3005–3018, 2020.

[26] S. Luo, P. Fan, K. Li, H. Xing, L. Luo, and H. Yu, "Fast parameter synchronization for distributed learning with selective multicast," in *IEEE International Conference on Communications (ICC)*, 2022, pp. 4775–4780.

[27] Q. Luo, J. He, Y. Zhuo, and X. Qian, "Prague: High-performance heterogeneity-aware asynchronous decentralized training," in *Proceedings of the 25th International Conference on Architectural Support for Pro-*

gramming Languages and Operating Systems (ASPLOS), 2020, pp. 401–416.

[28] I. Cano, D. Mahajan, G. M. Fumarola, A. Krishnamurthy, M. Weimer, and C. Curino, "Towards geo-distributed machine learning," *IEEE Data(base) Engineering Bulletin*, vol. 40, pp. 41–59, Dec 2015.

[29] Q. Yang, Y. Liu, T. Chen, and Y. Tong, "Federated machine learning: Concept and applications," *ACM Transactions on Intelligent Systems and Technology*, vol. 10, no. 2, pp. 1–19, jan 2019.

[30] "Fedai ecosystem," https://www.fedai.org/cases/ (Accessed date: Jun 6 2024).

[31] A. Gibiansky and G. Diamos, "Baidu allreduce," https://github.com/baidu-research/baidu-allreduce (Accessed date: Jul 13 2022), Feb 2017.

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

[33] J. Huang, P. Majumder, S. Kim, A. Muzahid, K. H. Yum, and E. J. Kim, "Communication algorithm-architecture co-design for distributed deep learning," in *Proceedings of the ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 181–194.

[34] 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.

[35] J. Kim, W. J. Dally, and D. Abts, "Flattened butterfly: A cost-efficient topology for high-radix networks," in *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, 2007, pp. 126–137.

[36] R. Rabenseifner, "Optimization of collective reduction operations," in *Proceedings of the Computational Science - ICCS 2004*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 1–9.

[37] X. Lian, W. Zhang, C. Zhang, and J. Liu, "Asynchronous decentralized parallel stochastic gradient descent," in *Proceedings of the 35th International Conference on Machine Learning (ICML)*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80, 10–15 Jul 2018, pp. 3043–3052.

[38] S. Luo, P. Fan, H. Xing, and H. Yu, "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.

[39] M. K. Aguilera, W. Chen, and S. Toueg, "Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks," *Theoretical Computer Science*, vol. 220, no. 1, pp. 3–30, jun 1999.

[40] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized task-aware scheduling for data center networks," in *Proceedings of the ACM SIGCOMM Conference*, 2014, pp. 431–442.

[41] P. Jain, S. Kumar, S. Wooders, S. G. Patil, J. E. Gonzalez, and I. Stoica, "Skyplane: Optimizing transfer cost and throughput using Cloud-Aware overlays," in *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 1375–1389.

[42] F. Lai, M. Chowdhury, and H. Madhyastha, "To relay or not to relay for Inter-Cloud transfers?" in *Proceedings of the 10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, Jul. 2018.

[43] G. Wang, S. Venkataraman, A. Phanishayee, N. Devanur, J. Thelin, and I. Stoica, "Blink: Fast and generic collectives for distributed ml," in *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze, Eds., vol. 2, 2020, pp. 172–186.

[44] L. Luo, S. Yang, W. Feng, H. Yu, G. Sun, and B. Lei, "Optimizing communication topology for collaborative learning across datacenters," in *Emerging Networking Architecture and Technologies*, W. Quan, Ed. Singapore: Springer Nature Singapore, 2023, pp. 184–197.

[45] R. Hong and A. Chandra, "Dlion: Decentralized distributed deep learning in micro-clouds," in *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2021, pp. 227–238.

[46] C. Zhang, S. Li, J. Xia, W. Wang, F. Yan, and Y. Liu, "BatchCrypt: Efficient homomorphic encryption for Cross-Silo federated learning," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, Jul. 2020, pp. 493–506.

[47] H. Pan, Z. Li, J. Dong, Z. Cao, T. Lan, D. Zhang, G. Tyson, and G. Xie, "Dissecting the communication latency in distributed deep sparse learning," in *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2020, pp. 528–534.

[48] "Mobilenet v3," https://pytorch.org/vision/stable/models/mobilenetv3.html (Accessed date: Jun 6 2024).

[49] H. Hsu, H. Qi, and M. Brown, "Measuring the effects of non-identical data distribution for federated visual classification," in *Neurips Workshop on Federated Learning*, 2019. [Online]. Available: https://arxiv.org/abs/1909.06335

[50] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding, "MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters," in *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2022, pp. 945–960.

[51] X. Zhao, A. An, J. Liu, and B. X. Chen, "Dynamic stale synchronous parallel distributed training for deep learning," in *Proceedings of the IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2019, pp. 1507–1517.

[52] C. Xu, G. Neglia, and N. Sebastianelli, "Dynamic backup workers for parallel machine learning," *Computer Networks*, vol. 188, p. 107846, 2021.

[53] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Computing Surveys*, vol. 52, no. 4, aug 2019.

[54] Q. Ma, Y. Xu, H. Xu, Z. Jiang, L. Huang, and H. Huang, "Fedsa: A semi-asynchronous federated learning mechanism in heterogeneous edge computing," *IEEE Journal on Selected Areas in Communications*, vol. 39, no. 12,

pp. 3654–3672, 2021.
[55] H. Xu, C.-Y. Ho, A. M. Abdelmoniem, A. Dutta, E. H. Bergou, K. Karatsenidis, M. Canini, and P. Kalnis, "Grace: A compressed communication framework for distributed machine learning," in *Proceedings of the IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, 2021, pp. 561–572.
[56] S. Luo, X. Yu, K. Li, and H. Xing, "Releasing the power of in-network aggregation with aggregator-aware routing optimization," *IEEE/ACM Transactions on Networking*, pp. 1–15, 2024.
[57] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo, "A generic communication scheduler for distributed dnn training acceleration," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019, pp. 16–29.
[58] W. Wang, M. Khazraee, Z. Zhong, M. Ghobadi, Z. Jia, D. Mudigere, Y. Zhang, and A. Kewitsch, "TopoOpt: Co-optimizing network topology and parallelization strategy for distributed training jobs," in *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2023, pp. 739–767.

Shouxi Luo (Member, IEEE) received the bachelor's degree in communication engineering and the Ph.D. degree in communication and information systems from the University of Electronic Science and Technology of China, China, in 2011 and 2016, respectively. He is currently an Associate Professor with Southwest Jiaotong University. His research interests include data center networks, software-defined networking, and networked systems.

Renyi Wang received the master's degree in computer science and technology from Southwest Jiaotong University, China, in 2023. His research interests include distributed deep learning and networked systems.

Ke Li received the Ph.D. degree in communication and information systems from the University of Electronic Science and Technology of China, China, in 2012. She is currently a Lecturer with Southwest Jiaotong University. Her research interests include machine learning, distributed systems, and the Internet of Things.

Huanlai Xing (Member, IEEE) received the B. Eng. degree in communications engineering from Southwest Jiaotong University, China, in 2006, the M. Eng. degree in electromagnetic fields and wavelength technology from the Beijing University of Posts and Telecommunications, China, in 2009, and the Ph.D. degree in computer science from the University of Nottingham, U.K., in 2013. Currently, he is an Associate Professor with Southwest Jiaotong University. His research interests include mobile edge computing, evolutionary computation, etc.