Dynamic Flow Scheduling for DNN Training Workloads in Data Centers

Xiaoyang Zhao*, *Student Member, IEEE,* Chuan Wu*, *Senior Member, IEEE,* and Xia Zhu†
*Department of Computer Science, The University of Hong Kong, Email: {xyzhao, cwu}@cs.hku.hk
† Huawei Technologies Co. Ltd., Email: zhuxia1@huawei.com

***Abstract*—Distributed deep learning (DL) training constitutes a significant portion of workloads in modern data centers that are equipped with high computational capacities, such as GPU servers. However, frequent tensor exchanges among workers during distributed deep neural network (DNN) training can result in heavy traffic in the data center network, leading to congestion at server NICs and in the switching network. Unfortunately, none of the existing DL communication libraries support active flow control to optimize tensor transmission performance, instead relying on passive adjustments to the congestion window or sending rate based on packet loss or delay. To address this issue, we propose a flow scheduler per host that dynamically tunes the sending rates of outgoing tensor flows from each server, maximizing network bandwidth utilization and expediting job training progress. Our scheduler comprises two main components: a monitoring module that interacts with state-of-the-art communication libraries supporting parameter server and all-reduce paradigms to track the training progress of DNN jobs, and a congestion control protocol that receives in-network feedback from traversing switches and computes optimized flow sending rates. For data centers where switches are not programmable, we provide a software solution that emulates switch behavior and interacts with the scheduler on servers. Experiments with real-world GPU testbed and trace-driven simulation demonstrate that our scheduler outperforms common rate control protocols and representative learning-based schemes in various settings.**

***Index Terms*—Machine Learning System, Networking for AI, Congestion Control Protocol.**

I. Introduction

Nowadays leading IT companies are managing machine learning (ML) clusters within data center environments. These clusters are utilized to execute deep learning (DL) jobs aimed at training ML models that cater to diverse business requirements. Modern ML models, such as deep neural networks (DNNs), have been rapidly growing in size. For instance, the latest language model, GPT-4 [1], boasts an impressive 1.76 trillion parameters. The scale of these large DNN models necessitates the adoption of distributed iterative training, involving multiple workers spread across multiple servers. During each training iteration, these workers exchange gradients and parameters with one another following local computation, resulting in significant network traffic and communication bottleneck. For example, in a scenario where a job trains a 1GB model with 1000 workers over 1000 iterations, the total traffic generated amounts to approximately 2PB [2].

Numerous approaches have been investigated to optimize communication performance in distributed DNN training. One strategy involves reducing the traffic volume associated with tensor transfers through techniques such as gradient quantization [3] or in-network aggregation [4]. Other approaches aim to maximize the overlap between tensor communication and gradient computation, for instance, through the utilization of wait-free backpropagation [5]. Additionally, careful transmission scheduling techniques, such as tensor partitioning/fusion [6] and transmission ordering [7], [8], have also been explored.

However, these existing approaches primarily focus on enhancing the training speeds of individual DNN jobs, neglecting the potential resource contention that arises when multiple jobs run concurrently in a shared cluster [9]. In such scenarios, numerous DL jobs are likely deployed on the same server, possibly sharing a single network interface card (NIC) and in-network links for transmitting tensor traffic. The communication libraries currently employed in distributed deep learning (i.e., NCCL [10], PS-Lite [11]) rely on conventional or OS-default rate control mechanisms for managing tensor flows. These protocols, such as CUBIC and Reno used in most kernels, are designed with rules to increase congestion windows for faster transmission and regulate congestion windows after congestion events (e.g., packet loss). In the context of distributed training, tensor traffic is generated periodically in bursts, with a large number of gradient packets transmitted within very short intervals following the completion of local computations in each training iteration. Reacting to congestion "after the fact" leads to inefficient transmissions and hampers the communication phases of jobs. Consequently, there is a lack of a proactive mechanism that can cooperatively interleave ML flows, mitigating the risk of network congestion and accelerating the training progress.

The networking community has a rich history of scheduling flows in data centers, ranging from individual flow scheduling [12], [13] to coflow scheduling [14]–[18] for application-level optimization. However, when it comes to distributed ML jobs, the coflow network abstraction falls short due to two main reasons. *First*, existing solutions typically consider two types of dependencies: "Start-After" (a coflow cannot start until another coflow is completed) and "Finish-Before" (a coflow cannot end until another coflow has ended). However, these dependencies do not meet the efficiency requirements of parameter synchronization in data-parallel training (we detail the unique dependencies of parameter server architecture and AllReduce collective in Sec. II-C). Scheduling flow rates without considering these dependencies can lead to bandwidth wastage and low throughput. *Second*, most coflow approaches

This work was supported in part by grants from Hong Kong RGC under the contracts HKU 17208920,17204423 and C7004-22G(CRF).

employ a centralized scheduler, prohibiting scalability of distributed ML. Instead of treating the entire data center as a large logical switch, each worker should monitor network statistics along the routing path, negotiate with dependent workers, and adjust its flow rate in a timely manner.

In this paper, we propose a distributed scheduling framework designed specifically for managing ML flows within a shared data center. Our scheduler runs in each server's user space and expedites job training for higher cluster throughput. It interacts with ML communication libraries to collect training status data, receives in-network feedback of link load from switches and schedules the transmission rate of tensor flows. Our contributions in developing the framework include:

▹ We design a cooperative mechanism between schedulers and switches. Each scheduler conveys lightweight messages about the current flow schedules to traversing switches, and the switches update load estimations of adjacent links over future time. By continuous information exchange, schedulers achieve global congestion-avoiding flow scheduling.

▹ We integrate a control protocol into each scheduler to compute the sending rates of flows at regular intervals. This protocol is derived from a networking problem of minimizing competition time among concurrent flows. It further takes into account the dependencies of ML flows and dynamically adjusts flow rates during each training iteration.

▹ We implement a prototype of our scheduler, which is compatible with the state-of-the-art communication libraries for run-time status collection and achieves individual control of each tensor flow sent out from the host. We also provide a software solution to support our framework with non-programmable switches.

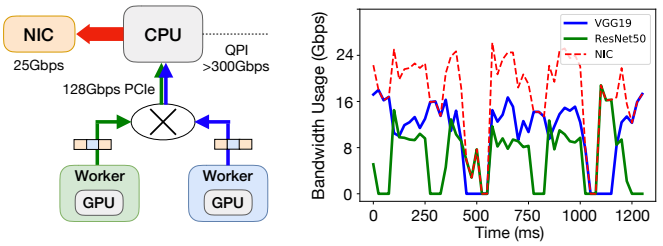
▹ We evaluate our system using representative DNN training workloads on a GPU testbed under various settings. The results demonstrate that our system can efficiently stagger bursty ML flows to avoid congestion, thus expediting training over 15% as compared to representative congestion control modules. Profiling results also indicate minimal overhead in scheduling latency and bandwidth consumption of message negotiations.

II. Background and Motivation

A. Distributed DNN Training

ML model training typically minimizes a loss function over a large dataset, which is time consuming. Many frameworks have been developed for distributed training, e.g., TensorFlow [19], MXNet [20] and PyTorch [21].

Synchronous data-parallel training is widely used in production DL workloads [8]. Each worker (e.g., one GPU device) computes gradients of model parameters using one mini-batch from the allocated partial training dataset, exchanges gradients with other workers, and updates the parameters of their local model replica. Gradient exchange among workers is typically achieved through a PS architecture [22] or an AllReduce algorithm [23]. With the PS architecture, one or multiple PSs aggregate gradients from all workers and send updated parameters back to the workers in each iteration. Using an AllReduce algorithm, workers perform a collective operation (e.g., ring collective [10]) to sum or average the gradients and disperse the aggregated gradients among themselves.



(a) Server Architecture (b) Bandwidth Consumption

Fig. 1: Congestion at server NIC due to burst ML flows, when two training jobs (VGG19 and ResNet50) are co-located in the same server with architecture shown in (a).

B. Bursty ML Flows Causes Congestion

Gradient/parameter tensors are transferred among workers for parameter synchronization, using different communication libraries with various DL frameworks. For example, MXNet uses PS-Lite [11], TensorFlow [19] uses gRPC, and PyTorch's DDP paradigm enables collective communication with different backends (e.g., Gloo [24], MPI [25] and NCCL [10]). However, none of these libraries allow in-network control of communication flows and rely on default congestion control algorithms in the kernel modules. In an RDMA environment, high-performance network stacks like Infiniband or RoCE are deployed [26], and conventional congestion control solutions (such as explicit congestion notification and priority-based flow control [26])) are still commonly used.

Training jobs may involve many workers located on different servers, leading to a large volume of network traffic. For example, a PS job training VGG16 with 10 workers can generate over 5GB of traffic in each iteration [27]. In addition, the bursty nature of tensor traffic during iterative training can cause contention at a server's NIC when multiple jobs/workers located on the same server are sending tensors concurrently. Fig. 1 shows a simple scenario of training VGG19 and ResNet50 models, concurrently, using ring all-reduce with Horovod and NCCL. Each server hosts a worker of each job, and each worker communicates with another worker of the same job on another server. The default CUBIC congestion control is used, and the bandwidth usage of each worker on the server is profiled using Nethogs and plotted in Fig. 1(b). As shown in the figure, communication from both jobs concurs over time, leading to congestion at the NIC and slower communication in both jobs. This inspires us to strategically adjust the transmission time and rate from different co-located DL jobs to manage bandwidth usage at NICs/network links for better overall job throughput.

C. Opportunities and Challenges

Opportunities exist with ML flow scheduling.

Periodic Training. In a shared data center, jobs arrive over time, and the duration of their computation phases varies due to differences in model complexity. Instead of pursuing optimal scheduling over the system's entire operation span, we practically aim to schedule concurrent flows in a periodic manner to reduce communication completion time.

We focus on a more common synchronous training scenario that involves numerous mini-batch iterations because asyn-

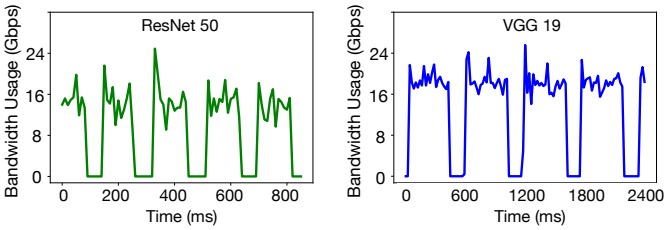
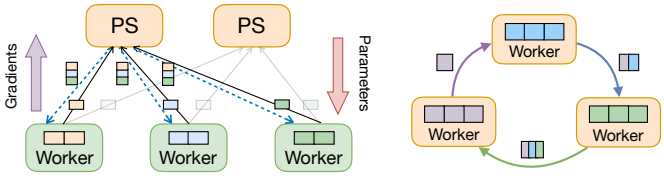


Fig. 2: Periodic communications of VGG19 and ResNet50.



(a) PS Architecture (b) AllReduce Scheme

Fig. 3: Parameter Synchronization Schemes.

chronous algorithms tend to trade off training efficiency for throughput by removing some of the iteration barriers.

Flow Dependency. During each iteration of a distributed job, tensor flows sent out from workers are interdependent. In the PS architecture depicted in Fig. 3(a), parameters can only be broadcast after aggregating the corresponding gradients received from workers. In the AllReduce collective, shown in Fig. 3(b), each worker must receive gradient chunks from the preceding worker, aggregate them with its own chunks, and send the result to the subsequent worker. Maintaining these dependencies is crucial to maximize throughput and efficiently utilize bandwidth during the communication phase. However, existing coflow solutions focus on the dependent relationship of start time or finish time among different transmissions, neglecting the dependencies during the whole period [28]. Considering the start time of different coflows (e.g., aggregation flows from workers to PSs as one coflow, broadcast flows from PSs to workers as the other coflow) leads to a low pipelining degree in transmission since the aggregated parameters could be immediately broadcast to all workers; on the other hand, if we focus on the finish time dependency, any flow rate allocation that violates the ML dependencies throughout the transmission can result in wasted bandwidth and inefficient transmission.

Improvement space exists by tuning the rates dynamically to cater to the dependency. Taking the PS job as an example, if any gradient flow sent to a PS is delayed, the scheduler on that PS node should reduce bandwidth allocation to the broadcast flows but assign higher transmission rates to other co-located jobs' flows. Conversely, considering that the PS has available bandwidth for sending broadcast flows, schedulers at workers should schedule faster gradient flow transmissions to best exploit the currently available bandwidth. Similarly, each worker in the AllReduce collective should have a lower allocated bandwidth if the transmission of its precedent worker is delayed, and take the opportunity to increase the transmission rate if its subsequent worker has a higher bandwidth.

Staggering Flows. ML training jobs alternate between computation and communication phases. When multiple ML flows compete for bandwidth on the same server or link, equal sharing of bandwidth can lead to reduced transfer speeds and congestion due to bursty traffic patterns. In the worst-case scenario, multiple jobs undergo communication phase simultaneously, intensifying the competition for bandwidth. When they step into the following computation phase, the network bandwidth remains under-utilized. Hence, staggering flows is an effective method to alleviate congestion, enabling each transmission to fully utilize the available bandwidth and facilitating overlapping of computations and communications of differnet jobs. For example, consider two co-located training jobs, ResNet50 and VGG19. If the ResNet50 job is in its communication phase while the VGG19 job completes a computation phase and is ready to transmit gradients, staggering the communication in the VGG19 job until ResNet50 completes its communication allows the ResNet50 job to commence its next computation phase earlier, potentially overlapping with the communication phase of VGG19.

Determining the optimal flow staggering schedule for multiple jobs in a shared cluster is challenging, particularly when individual host schedulers make independent decisions. Host schedulers have limited visibility for decision-making in two key aspects. *Firstly*, they lack awareness of the workloads on in-net links that their flows traverse, as these links are shared by flows from different hosts. This makes it difficult to regulate flow rates before congestion occurs within the network. *Secondly*, the scheduling strategies employed by other schedulers are not disclosed, posing obstacles to global scheduling. To address these challenges, our design incorporates switches to provide schedulers with information on future load over routing paths. The scheduler adopts an efficient rate control solution derived from a global optimization problem.

In summary, none of the existing scheduling systems could satisfy the aforementioned requirements to accelerate communication in ML jobs.

III. System Framework

We consider a data center consisting of GPU servers interconnected by a switching network. Data-parallel DNN training jobs are submitted to run in the data center over time, which use either the PS architecture [22] or the all-reduce paradigm [23] for gradient/parameter communication. Workers (PSs) of a job could be assigned to one or multiple servers, due to resource fragmentation. Our scheduling framework consists of three key modules as shown in Fig. 4.

A. A scheduler on each server host

We deploy a scheduler on every server hosting distributed training jobs. Each scheduler runs in the user space of the host as an independent process. Upon the arrival of a training job, the scheduler on each of the servers, where a worker or PS of the job is deployed, collects the job information including the architecture used for parameter synchronization and the total parameter size of the model (which can be specified by the job owner). During training, the *monitoring block* in each scheduler interacts with the communication library of the worker/PS to track the tensor transfers by comparing the volume of delivered tensors and the total model gradients.

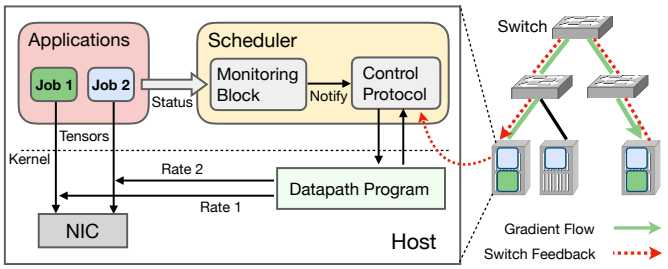


Fig. 4: System Framework Overview.

Each scheduler works in a time-slotted manner. When a tensor flow from a worker/PS starts, the scheduler determines the sending rate of the flow in each time slot, based on a carefully designed *control protocol*, until the tensors are fully transmitted. The determined sending rates are passed to the *datapath program* launched in the kernel to control actual packet transmissions and will not change within one time slot.

Monitoring Block. It interacts with our customized communication libraries that each worker/PS uses. To achieve individual flow control, the monitoring block identifies different connections between workers or PSs with their socket handlers, collects the size of tensors ready to be synchronized from the respective communication module and infers the job training status, i.e., which iteration the training has reached, whether the communication phase has started in the current training iteration and how many tensors remain to be transmitted for the current flow. It only focuses on inter-server flows (distinguished according to source and destination IP addresses), since intra-server communication is much faster through NVLinks. The training status information will be exploited by the control protocol to dynamically adjust the sending rates of flows.

Control Protocol. The protocol receives notifications from the monitoring block at the start of each new communication round of a training job. It includes the newly generated flows in its scheduling process. These flows follow the ECMP routing protocol [29] for their traversal. The scheduler retrieves link load estimations over future time slots from switches (**Sec. IV-A**) and determines the initial sending rate at the beginning of each time slot (**Sec. IV-B**).

Then over time, the scheduler dynamically adjusts the flow transmission rates, according to link load dynamics and flow dependencies (**Sec. IV-C**). The goal is to reduce the communication time in each iteration of each training job.

B. Modified communication libraries

We modify DL communication libraries (NCCL [10] for all-reduce jobs and PS-Lite [11] for PS jobs) to allow the communication module of a worker/PS to interact with our scheduler located on the same server. When tensors are ready for communication, the accumulative size of tensors enqueued into CUDA kernel (in NCCL) or pushed/pulled with remote servers (in PS-Lite) is recorded and stored into host shared memory. The monitoring block in the co-located scheduler can then fetch the value to infer the training status. We detail the implementation in **Sec. V**.

C. Switch

To prevent congestion on in-network links, our approach assumes the presence of programmable switches in the data center. These switches are capable of estimating future loads on adjacent links for a specified number of time slots (10 time slots in our setting). Each switch collects schedule information carried in bypassing flows, including planned transmission rates for future time slots. By aggregating the traffic load for each link across future time slots, the switch obtains load estimations. These load estimations are then sent to the schedulers on servers as in-network feedback, which enables the schedulers to anticipate potential congestion caused by concurrent training jobs and make more effective plans for tensor transfers. In data centers where programmable switches are not available, we provide a software-based solution that serves as a switch module on servers (refer to **Sec. IV-A**).

IV. Detailed Design

In Sec. IV-A, we first present how switches provide estimated load information to schedulers. Following that, in Sec. IV-B, we formulate a global flow scheduling problem and employ an online primal-dual algorithm to derive the initial schedule, leveraging the provided load information as input. Finally, in Sec. IV-C, we describe the detailed control protocol implemented in our distributed schedulers. This protocol dynamically adjusts flow rates to cater to flow dependencies, based on the initial schedule.

A. Switch module

Link Load Estimation. Each switch maintains a load table recording the estimated link load over future K time slots (including the current time slot when the flow schedule is being made), as shown in Fig. 5. Each traversing flow carries its transmission schedule in the packet payload, which is a set of rates over future K time slots decided by the sender-side scheduler. We assume the flow routing path is decided by ECMP protocol [30] and keeps unchanged. Switches extract bypassing flows' schedule payload and update load estimation of their adjacent links by accumulating them with scheduled rates of other bypassing flows in each future time slot.

For example in Fig. 5, a flow carrying its rate schedule (i.e., 5Gbps in time slot t_1, 20Gbps in t_2, etc.) traverses links l_1 and l_4 through a switch. In the switch's load table, rates of the flow (marked in red) are then added onto the original load estimation to render the new load estimation in rows corresponding to the links.

Link Load Feedback. Along the path of a flow, each bypassing switch encodes the load estimation (i.e., a K-element array) of its outgoing link in the path into the payload of a few packets of the flow. The load of links in the flow's path in the same time slot is summed up to indicate the overall *path load* of transmitting the flow in the time slot. The *path load* is carried in the flow packet and the flow receiver will encode it into the payload of ACK packets sent back to the sender of the flow. To allow schedule information exchange among schedulers, the receiver of a flow also encodes the transmission status of its outgoing flows (if any) into the payload of ACK

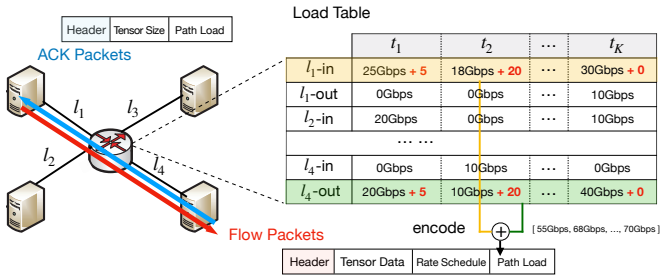


Fig. 5: Illustration of the load table in a switch module.

packets sent back to the flow sender, including sizes of tensors transmitted. The load estimation as well as the transmission status can then be collected from the corresponding ACK packets, which will be exploited by the schedulers.

Software Implementation. Switches are required to be programmable to achieve aforementioned functionalities, such as supporting P4 [4] protocol or being equipped with an FPGA board [2]. To implement data storage in a load table format within the switches, a key-value pair structure could be utilized. Each entry in the table would consist of a unique key, representing the specific time slot, and a corresponding value, storing the accumulated link load values. The switches could also be programmed to extract the schedule contents carried in the payload of packets and update the load table accordingly.

However, most data center consists of conventional switches with only packet forwarding capabilities. Thus, we provide a software solution that emulates the additional switch behavior. For each non-programmable switch, we execute a background process on a server to simulate the switch. This switch process is notified of the corresponding switch's link connectivity and maintains the load table similarly to a programmable switch. The schedule and link load estimations are explicitly exchanged between schedulers and emulated switches. A scheduler transmits messages containing its flow schedules to the emulated switches to update link load estimations, requests link load estimations from emulated switches and receives the response messages. Since the message sizes are significantly smaller than tensor sizes, the message passing time and bandwidth consumption between schedulers and emulated switches are negligible (as assessed in Sec. VI-G).

B. Basic Flow Scheduling Problem for Each Job

Original Global Problem. We first formulate the offline centralized problem to minimize all flows generated by concurrent jobs (tentatively not considering flow dependencies). Denote J as the job set, N_j as the flow set of job $j \in J$:

$$\text{minimize} \sum_{j\in J} \sum_{f\in N_j} \beta_f \tag{1}$$

subject to:

$$\forall j \in J, \forall f \in N_j : \beta_f = max_{t\in[T]}\{t | r_f(t) > 0\} \tag{2}$$

$$\forall j \in J, \forall f \in N_j : \hat{t} \sum_{t=1}^{T} r_f(t) \geq G_f \tag{3}$$

$$\forall l \in L, \forall t \in [T] : \sum_{j\in J} \sum_{f\in N_f} r_f(t) \leq u_l(t) \tag{4}$$

$$\forall j \in J, \forall f \in N_j, \forall t \in [T] : r_f(t) \geq 0 \tag{5}$$

where T is the system time horizon ($[T] = 1, ..., T$), $r_f(t)$ is the decision variable denoting the sending rate of flow f, $\hat{t}$ is the duration of a time slot, G_f is the transmitting size of flow f and $u_l(t)$ is the available bandwidth capacity of link $l \in L$ in time slot t (link capacity excluding non-ML cross traffic at t). In this problem, constraint (2) defines the completion time of each flow, constraint (3) ensures that the gradient tensor is successfully transmitted and constraint (4) is the link capacity.

Decomposition with Switch Feedback. Instead of adopting a centralized approach that assumes complete knowledge of all jobs and links, we propose decomposing the problem into individual jobs. In problem (1), variables related to different jobs are only interconnected by constraint (4). However, the impact of concurrent jobs on link load can be reflected through the link load estimation obtained from switches, as described in Sec. IV-A. Additionally, ML jobs are executed in an iterative manner, with each iteration following the next. Therefore, we assign schedulers the task of minimizing the makespan of the flow set (represented as $M_{i,j}$) generated during iteration i of job j. To circumvent the non-conventional constraint (2), i.e., non-linear *max* operator, we denote a valid variable s as a set of rates $\{r_f(t)\}_{f\in M_{i,j}}$ over future K time slots (since the switch module maintains the load estimations of those time slots) and satisfies the constraints (3)(5). Let $S_{i,j}$ be the set of feasible schedules for flow set $M_{i,j}$, the original problem is converted to the following ILP to identify the best schedule $s \in S_{i,j}$ which minimizes the communication makespan of iteration i of job j:

$$\text{minimize} \sum_{s\in S_{i,j}} x^s_{i,j} \beta^s_{i,j} \tag{6}$$

subject to:

$$\forall t = [T], \forall l \in L : \sum_{s\in S_{i,j}} x^s_{i,j} \sum_{f:l\in L_f} r^s_f(t) + b_l(t) \leq u_l(t) \tag{7}$$

$$\sum_{s\in S_{i,j}} x^s_{i,j} = 1 \tag{8}$$

$$\forall s \in S_{i,j} : x^s_{i,j} \in \{0, 1\} \tag{9}$$

where $x^s_{i,j}$ is the binary decision variable indicating whether schedule s is chosen ($x^s_{i,j} = 1$) or not ($x^s_{i,j} = 0$), L_f is the routing path of flow f, and $b_l(t)$ is the estimated load on link l in time t by the adjacent switch module, i.e., the total traffic of other concurrent jobs on l in t. Given a feasible schedule s, flow rate $r^s_f(t)$ for each flow in each time slot and the communication completion time $\beta^s_{i,j}$ are both determined.

Convert into Dual Problem. Problems (6) and (1) are equivalent because they share the same objective and feasible solutions. However, the decision variable associated with s can grow exponentially and flows typically start at different times due to the distributed placement of workers. Thus, we design an algorithm to derive the best rate allocation for each flow upon it being ready, exploiting the primal-dual framework to solve it in polynomial time.

We first formulate the dual of (6) by relaxing integrity constraint (9) and associate dual variables $\lambda_l(t)$ and $\nu_{i,j}$ with constraints (7) and (8), respectively:

$$\text{maximize} \quad \nu_{i,j} - \sum_{l \in L} \sum_{t \in [T]} \lambda_l(t)\big(u_l(t) - b_l(t)\big) \tag{10}$$

subject to:

$$\forall s \in S_{i,j} : \nu_{i,j} \le \beta^s_{i,j} + \sum_{f \in M_{i,j}} \sum_{t \in [1,T]} r^s_f(t)\Big(\sum_{l \in L_f} \lambda_l(t)\Big) \tag{11}$$

the dual variable $\lambda_l(t)$ can be interpreted as the unit bandwidth cost on link l in t, then the RHS of (11) can be regarded as the total transmission cost of adopting schedule s for the current communication flows. By the rule of complementary slackness [31], variable $x^s_{i,j}$ equals 0 unless the constraint (11) in its dual problem is tight. Thus, in order to minimize the dual objective, the chosen schedule s^* should hold the following, while satisfying feasibility constraints (3)(5):

$$s^* = \arg\min_{s \in S_{i,j}} \text{RHS of (11)}$$

Solve in Polynomial Time. To minimize the RHS of (11), we observe that when we enforce a time $\hat{\beta}_{i,j}$, before which the communication phase needs to be complete, the best rate schedule s should have the minimal total transmission cost. We obtain it by greedily assigning sending rates $r_f(s)$ of each flow to time slots with the least cost $\{\sum_{l \in L_f} \lambda_l(t)\}$ until all its tensor volume G_f can be transmitted. This is equivalent to allocating rates to time slots with lower path load retrieved from switch modules (Sec. IV-A), because the transmission cost increases with the accumulation of traffic load, and heavier traffic load induces larger transmission costs [31]. We use the sum of link load feedback accumulated along the routed switches as the path load for rate allocation. The rationale behind this is to prevent concurrent flows from causing traffic bursts at the same time on the same routing link, thereby staggering flows temporally and avoiding potential congestion and bottlenecks.

The sketch of our algorithm is shown in Alg. 1. We iterate over the communication completion time slots $\hat{\beta}_{i,j}$ from 1 to T (line 1): for each $\hat{\beta}_{i,j}$, we use a greedy method to allocate the sending rates of the flow f over time slots in the range $[1, \hat{\beta}_{i,j}]$. We begin by sorting time slots based on the transmission path load, in non-decreasing order, and store the sorted time slots in a set H (line 2). As long as there are remaining tensors to be transmitted and available time slots for allocation (line 4), we select the first time slot τ from set H (line 5) and allocate the flow rate in a best effort manner (line 6). The assigned rate for flow f at any time t should not exceed the minimum available bandwidth along its routing path, which is denoted as $\min_{l \in L_f} u_l(t) - b_l(t)$. Since non-ML flows are not controlled by our schedulers, the host side will estimate the minimal available bandwidth that an ML flow could achieve (under the prerequisite that the performance of non-ML flows is not compromised) as the difference between the maximal recorded load on the link over a past time window (4 time slots in our experiments) and the estimated link load caused by other flows in the respective future time slot.

After allocation, we update the remaining tensor volume and transmission cost accordingly (line 7-8), and remove the allocated time slot from set H (line 9). This process repeats until all required tensors are transmitted or there are not enough available time slots for allocation. In the former case, we compare the transmission costs induced by schedules enforced at different times and select the best schedule s^* that achieves the lowest transmission cost (line 10-13).

Algorithm 1: Greedily Scheduling Flow $f \in M_{i,j}$

```
Input: L_f, G_f, Σ_{l∈L_f} λ_l(t), u_l(t), b_l(t)
Init: r_f(t) = 0, H = ∅, s* = ∅, o = ∞
1  for β̂_{i,j} = 1 to T do
2      Sort time slots in [1, β̂_{i,j}] according to Σ_{l∈L_f} λ_l(t)
         in non-decreasing order into set H.
3      Initialize g = G_f, cost = 0.
4      while g > 0 and H ≠ ∅ do
5          Set τ to be the first element in H.
6          r_f(t) = r_f(t) + min_{l∈L_f} u_l(t) − b_l(t)
7          g = g − r̂_f(t)
8          cost = cost + Σ_{l∈L_f} λ_l(t)
9          Remove τ from H.
10     if g ≤ 0 then
11         õ = β̂_{i,j} + cost
12         if õ < o then
13             o = õ, s* = {r_f(t)}
   Return: s*
```

The algorithm can be executed in a distributed manner by different schedulers as the guidance of flow control. We next describe the detailed behaviors of a scheduler, which dynamically adjusts flow rates to cater to dependency requirements.

C. Distributed Schedulers

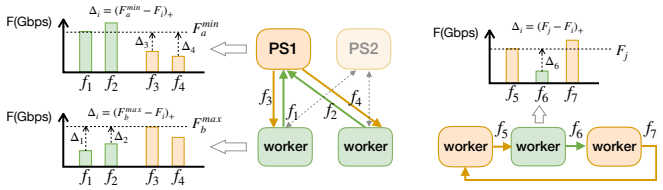
A scheduler schedules flows of a job (located on the same server as the scheduler) when the job has completed the computation phase in a training iteration. The goal is to collectively minimize the communication makespan in each training iteration of the training jobs through distributed scheduling.

Control Protocol. At the beginning of a time slot, a scheduler allows a few packets of each flow to be transmitted at the line rate, in order to retrieve the link load estimation piggybacked in ACK packets. Once the load estimation returns, the scheduler computes the initial rate schedule following Alg. 1: with the link load estimation over future K time slots, it assigns higher sending rates to time slots with lower traffic, until all tensors in the flow can be transmitted.

After computing the initial schedule of flow rates in the next K time slots (starting from the current time slot), the scheduler adjusts the flow transfer rate of the current time slot, catering to flow dependencies according to allocated rates of other flows in the same training job. Fig. 6 illustrates how the scheduler calculates the rate adjustment in the two cases of using PS or all-reduce for parameter synchronization, respectively.

PS Architecture. At any time slot of a PS, the size of tensors carried by a broadcast flow should always not be larger than the size of successfully aggregated gradients by each aggregation flow. The dependency is formally presented as:

$$\forall p \in P_j, t \in [T] : \min_{f \in a_p} \sum_{\tau \in [t]} \hat{tr}_f(\tau) \ge \max_{f \in b_p} \sum_{\tau \in [t]} \hat{tr}_f(\tau)$$



(a) PS (b) All-reduce

Fig. 6: Illustration of how a scheduler caters to flow dependency under PS architecture and all-Reduce paradigm.

where P_j is the set of PS of job j, a_p is the set of flows to ps p for aggregation, and b_p is the set of flows broadcast from ps p to workers. Take Fig. 6(a) as an example, a training job uses 2 PSs and 2 workers, each located on a dedicated server. $PS1$ receives aggregation flows f_1 and f_2 from the two workers, and sends broadcast flows f_3 and f_4 to the workers, respectively. The total size of each flow transmitted from the flow start till the end of the current time slot can be computed by summing up the products of the time slot length (60ms in our experiments) and the flow rate in each time slot (actual flow sending rate in an earlier time slot and the initially scheduled rate for the current time slot), across all time slots. We use F_i to denote the accumulated rate of flow f_i (sum of actual sending rates in previous time slots and the initially scheduled rate for the current time slot), use F_a^{min} to denote the minimal accumulated rate among aggregation flows at a PS (f_1 and f_2 in the example) and use F_b^{max} to represent the maximal accumulated rate among broadcast flows at the PS (f_3 and f_4 in the example). The scheduler at the worker that sends an aggregation flow f_i to the PS increases the flow rate of the current time slot by $\Delta_i = \max\{F_b^{max} - F_i, 0\} = (F_b^{max} - F_i)_+$, to better exploit potentially available outbound bandwidth at the PS. On the other hand, the scheduler at a PS that sends a broadcast flow f_i increases the rate by $\Delta_i = (F_a^{min} - F_i)_+$, to cater to available outbound bandwidth from the worker nodes.

All-reduce Paradigm. For each worker sending out a flow f, at any time slot, the tensor size sent to the subsequent worker should be no larger than the tensor size received from the precedent worker. Formally:

$$\forall t \in [T] : \sum_{\tau \in [t]} \hat{tr}_{C_f}(\tau) \geq \sum_{\tau \in [t]} \hat{tr}_f(\tau)$$

where C_f is the flow sent out from the precedent worker of flow f. Take Fig. 6(b) as an example, 3 workers synchronize the same gradient chunk in a ring through flows f_5, f_6, and f_7. The scheduler at the worker that sends flow f_i increases the flow rate by $\Delta_i = (F_j - F_i)_+$, where F_j represents the accumulated rate of precedent flow j (e.g., f_5 is the precedent flow of f_6), in order to match the potentially higher rate in chunk synchronization.

In our flow rate adjustment, we attempt to increase flow rates to meet flow-dependency expectations as well as to use maximally possible rates for most expedited communication. The rate to increase for a flow f_i may not be achievable due to bandwidth limitation along the flow path. The respective scheduler further estimates the maximal available bandwidth B_i to send a flow according to the maximum flow sending rate

Algorithm 2: Rate Allocation of Flow f_i

Input: B_i, link load estimation over K time slots
1 Infer remaining tensor size G to be transmitted in current iteration.
2 Compute *initial schedule* with Algorithm 1.
3 Set r as the initial rate derived for *initial schedule*.
4 $\Delta_i \Leftarrow computeRateAdjustment()$.
5 **if** $\Delta_i = 0$ **then**
6 Apply rate r.
7 **else**
8 **if** $\Delta_i > 0$ *and* $r + \Delta_i \leq B_i$ **then**
9 Apply rate $(r + \Delta_i)$.
10 **else if** $r + \Delta_i > B_i$ **then**
11 Apply rate B_i.
12 Greedily adjust future transmission rates of f_i based on link load estimation.
13 **if** *receiver of* f_i *is a PS in a PS job* **then**
14 Notify the PS to reduce rates of its broadcast flows j's by $(F_j - F_a^{min})_+$.
15 **if** *receiver of* f_i *is a worker in an all-reduce job* **then**
16 Notify the worker to reduce rate of its subsequent flow f_j by $(F_j - F_i)_+$.
17 Encode adjusted rate schedule into packet payload.
18 **function** *computeRateAdjustment()*:
19 **if** f_i *is an aggregation flow in a PS job* **then**
20 $\Delta_i = (F_b^{max} - F_i)_+$
21 **if** f_i *is a broadcast flow in a PS job* **then**
22 $\Delta_i = (F_a^{min} - F_i)_+$
23 **if** f_i *is a flow in an all-reduce job* **then**
24 Depend on the precedent flow f_j of f_i,
25 $\Delta_i = (F_j - F_i)_+$
 Return: Δ_i

over a past time window (4 time slots in our experiments), following the idea of BBR algorithm [32], and limits the flow sending rate of the current time slot accordingly. The scheduler obtains the actual sending rates by collecting transmission statistics from the kernel.

The complete algorithm for a scheduler to decide the actual sending rate of its flow f_i in the current time slot is given in Alg. 2. When the increased rate Δ_i equals 0 (line 5), indicating the flow dependency is satisfied, the actual applied rate in the current time slot follows that derived in the initial schedule. In case increasing the initial rate by Δ_i is achievable (line 8), the scheduler uses the increased rate and adjusts future time slots' transmission rates of the flow accordingly, using the same greedy approach as setting the initial rate schedule. Otherwise (line 10), the scheduler sets the flow rate to B_i and adjusts future rates accordingly. After the rate adjustment, the scheduler notifies the flow receiver's scheduler to adjust allocated rate(s) of its subsequent flow(s) (if any), if the latter's allocated rate(s) is/are larger than the predecessor flow's (lines 13-16): (i) for a PS receiver in a PS job, rate of each broadcast flow f_j is reduced by $(F_j - F_a^{min})_+$; (ii) for a worker receiver in an all-reduce job, its subsequent flow f_j's rate is reduced

by $(F_j - F_i)_+$. The released bandwidth can then be utilized for other co-located jobs' flows. The adjusted rate schedule is encoded into the payload of outgoing packets, which will be extracted by each traversing switch to update their load estimations, as described in Sec. IV-A.

Time Complexity Analysis. When executing the rate allocation Alg. 2, it takes $O(T^2logT)$ to compute the initial schedule with Alg. 1: enumerating the completion time $\hat{\beta}_{i,j}$ takes $O(T)$ and sorting time slots according to their costs takes $O(TlogT)$ time. The rest part of Alg. 2 takes constant time, then the overall time complexity is $O(T^2logT)$. We further evaluate the scheduling overhead in Sec. VI-G.

Compatibility with legacy protocols. The control protocol run by our schedulers is compatible with mainstream congestion control protocols. Our schedulers only enforce sending rates of ML flows in a data center, with packet structure and semantics remaining the same as in TCP (ACK, hand-shaking, re-transmissions, etc.). Non-ML flows in the data center, which use TCP as the de-facto congestion control protocol, are not affected by our protocol.

V. Prototype Implementation

We implement a prototype system using emulated switches, which is open-sourced at *https://github.com/joeyyoung/mlcc*.

Scheduler with Kernel Support. Our scheduler employs the CCP architecture [33], which is an off-datapath congestion control plane and uses user-space signals to control the datapath's transmission rate. A kernel module is launched in the Linux OS (kernel version 5.4.0), within which a compatible datapath program enforces the rate control algorithm specified by our user-space scheduler. The datapath program is event-driven and developed in LISP-like syntax, and runs in the context of each individual flow. In every time slot (60ms in our implementation), the datapath program returns both ACK-level and flow-level measurements, e.g., recent sampled minimal round-trip time (RTT), outgoing sending rates, etc.

Communication Library Customization. To enable interaction with our scheduler's monitoring block and enforce our control protocol in flow transmission, we slightly modify the communication libraries that are commonly used by distributed training frameworks.

We add support in the NCCL library [10] for all-reduce jobs with patches. For ring all-reduce that we focus on, GPUs located on different servers communicate with each other using the CPU $Proxy$ thread launched at both the sender and the receiver. In the TCP environment, we set the socket option of $Proxy$ to adopt our control protocol. During initialization, NCCL bootstraps the ring formulation by collecting IP addresses from all ranks (workers) and then broadcasting information of the previous and next ranks to each worker. We filter out those bootstrapping flows which are not controlled by our scheduler. During training, tensors ready for communication are enqueued into the CUDA kernel with $ncclEnqueueCheck()$, and the accumulated tensor size in each iteration is recorded into shared memory. The local monitoring block compares the size of transferred tensors and the total model size, to identify the training progress and determine the remaining tensors to be transmitted.

We also add patches to the PS-lite library [11] which is commonly used by frameworks such as MXNet [20] and BytePS [22]. In the low level, PS-lite uses the ZeroMQ API [34] for data transmission. We set the TCP socket option to enforce the use of our control protocol when the zmq socket is created. During training, PS-lite tracks data push/pull with remote nodes and interacts with the monitoring block in the same way as what we do in NCCL.

No modifications of the training script and the original DL framework (e.g., PyTorch [21], MXNet [20]) are needed.

Emulated Switch. Each emulated switch is implemented as a daemon process with two gRPC services, invoked in each time slot. (1) $fetch\_feedback()$: a scheduler sends requests to the switch, and the switch responds with the load estimation of future K time slots (we set $K = 10$). (2) $update\_schedule()$: the scheduler sends a message containing newly computed flow schedule to by-passing switches, and the switches update their link load estimations and respond with a status code indicating success or not.

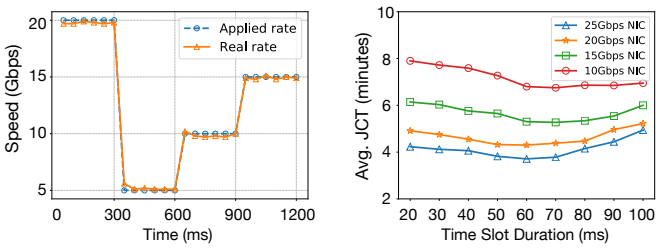
VI. Physical Testbed Evaluation

A. Methodology

Testbed. We conduct experiments in a cluster of 6 servers connected by a Dell Z9100-ON switch. Each server is equipped with two GTX 1080Ti GPUs, one 8-core Intel E5-1660 CPU, 48GB RAM and one MCX413A-GCAT NIC with peak bandwidth 25Gbps. We install Ubuntu 18.04.5 LTS (Linux Kernel 5.4.0-77-generic) with NVIDIA GTX driver 455.45.01, CUDA 10.2 and CuDNN 7.6.5 on each server.

DNN training jobs. We run jobs training CNN models, VGG19 (549 MB, using all-reduce), VGG16 (528MB, using PS) and ResNet50 (98MB, using PS) on down-scaled ImageNet dataset, and transformer model, GPT2 (1.2 GB, using all-reduce) on down-scaled person-chat dataset [35]. We use Horovod 0.22 with NCCL 2.11.4 to run ring all-reduce jobs and BytePS 0.2.4 for PS jobs, and jobs with the two patterns can coexist on the same server. All training scripts are written with PyTorch [21] and from the official repositories [22] [23]. Jobs are submitted following the job arrival pattern (i.e., the number of arrived jobs per minute) extracted from Google traces [27], which contains lifetime information and resource consumption of each job managed by Google cluster management software internally known as Borg. We down-scale the arrival rates for a reduced workload scale. Each PS job uses 2 parameter servers and 2 workers; each all-reduce job uses 3 workers. Each worker occupies 1 GPU and the batch size per GPU is 64. Jobs are queued if GPUs are not available, and whenever a job completes training, the queued job will be launched immediately.

We inject cross-traffic flows among the 6 servers, which use default TCP congestion control protocol in the kernel. Each server randomly chooses a destination to inject a non-ML flow by iperf every 2 seconds, and each flow lasts 0.2 to 2 seconds. We use the average job completion time (JCT) to evaluate the throughput of cluster, and the training speed (images/sec for



(a) Rate Consistency (b) Effect of Time Slot Duration

Fig. 7: Rate consistency and time slot configuration.

CNNs and tokens/sec for the Transformer) as the job performance metric, which is computed by $\frac{\text{batch size}}{\text{training time per iteration}}$. Reported speed numbers are averaged over 100 training iterations.

Baselines. We compare our scheduler system with various flow/congestion control protocols:

- TCP Variants. We choose several commonly used TCP algorithms: (1) CUBIC [36], the default congestion control protocol since Linux kernel version 2.6, with a CUBIC window growth function; (2) Reno [37], which applies the AIMD method to control the congestion window with fast retransmission and recovery mechanisms; (3) BBRv1 [32], available since Linux kernel version 4.9, which periodically estimates the available bandwidth and round trip time (RRT, i.e., the actual time taken for a packet to travel from the source to the destination and then back) to reach the maximum throughput; (4) DCTCP [38], supported as a module in Linux, which relies on explicit congestion signal (ECN) to adjust sending window.
- PCC [39]: a well-known performance-oriented protocol. It tries sending at two different rates, and moves in the direction that empirically results in greater performance utility. We use the user-space implementation as the baseline.
- Aurora [40]: a deep reinforcement learning (DRL)-driven congestion control scheme. The input to the sender agent contains a fixed-length history of measured statistics, and its actions are translated to changes in sending rates in each time interval. We use traces collected in our cluster to train the DRL model, and deploy the trained model online for inference of rate adjustments.
- DeepCC [41]: it adopts a DRL agent which takes measurements (loss, RTT, throughput) as input and produces a sending rate. We train the model offline and conduct online inferences to determine sending rates periodically (100 ms).

None of the above algorithms consider the unique traffic patterns and dependencies of ML jobs.

B. Rate Consistency and Time Slot Configuration

We first examine the consistency between the sending rate computed by the scheduler and the actual flow rate profiled. We use iperf to generate a long lasting flow between two servers, and use the scheduler to enforce an explicit rate in every 50ms. As Fig. 7(a) shows, our user-space scheduler can precisely control the flow rate. Even in cases where the desired sending rates vary greatly, such as from 20 Gbps to 5 Gbps at the time of 300 ms, it is possible to control the actual transmission as expected.

The time slot duration is a crucial factor in determining the scheduling granularity when utilizing Alg. 1 to derive an initial schedule for each flow. It also influences the duration of negotiation with emulated switches and the evaluation of dependency fulfillment for dynamic adjustments. A smaller time slot duration enables more fine-grained flow rate control and allows for timely adjustments to address potential congestion and dependency violations. This finer granularity facilitates optimized network performance by promptly responding to changing conditions. However, as the time slot duration decreases, the proportion of negotiation and computation delays relative to the overall duration increases (evaluated in Fig. 12). Consequently, this can impact the precision and effectiveness of the scheduling decisions made by the algorithm. To quantify this phenomenon, we launch benchmark jobs as described above and record the average JCT under different NIC speeds (using Linux tc tool) and time slot durations as shown in Fig. 7(b). For the latter experiments, we empirically choose 60ms as the time slot duration that achieves maximum system throughput.

C. System Throughput

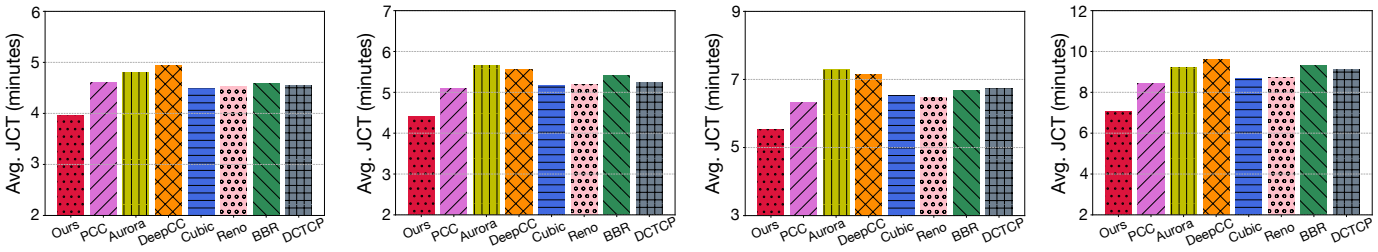
Figure 8 presents a performance comparison between our schedulers and baselines at various bandwidth levels. The results demonstrate at least 15% reduction in average JCT compared to TCP variants. These passive congestion control algorithms perform similarly and fail to consider the bursty nature of ML flows, i.e., only when the congestion signal is detected (loss packet or explicit signal), the sending window size decreases and grows until the next congestion occurs. Additionally, the dependency on flows sent out from different workers/PSs is ignored, making the overall communication inefficient. Table I provides additional insights into the observed JCT by reporting the minimum, maximum, and 99th percentile values from the trace. Our protocol staggers concurrent flows to enable the overlapping of computation and communication phases for different jobs and iteratively optimizes the communication makespan of each iteration to eliminate any job starvation.

Our protocol also outperforms Google's BBR algorithm. As a proactive control mechanism that probes the maximum bandwidth and the minimum RTT, BBR suffers from the convergence problem with limited observations of the sender: during the transmission of a BBR flow, when other ML flows burst and compete for the bandwidth, it will take BBR multiple probing rounds to decrease the flow's sending rate to an appropriate value, causing serious congestion. PCC experiences the same problem as BBR, and heavily relies on trying different sending rates to achieve the best bandwidth utilization. DNN training alternatively carries out computation and communication, and the communication time in one iteration is respectively small. Hence, there is not enough chance for a PCC flow to try numerous sending rates and choose the best one from them.

As compared to learning-based algorithms Aurora and DeepCC, our scheduler is more than 28.9% better in terms of system throughput. The poor performance of learning-based control is due to the dynamic network environment, which

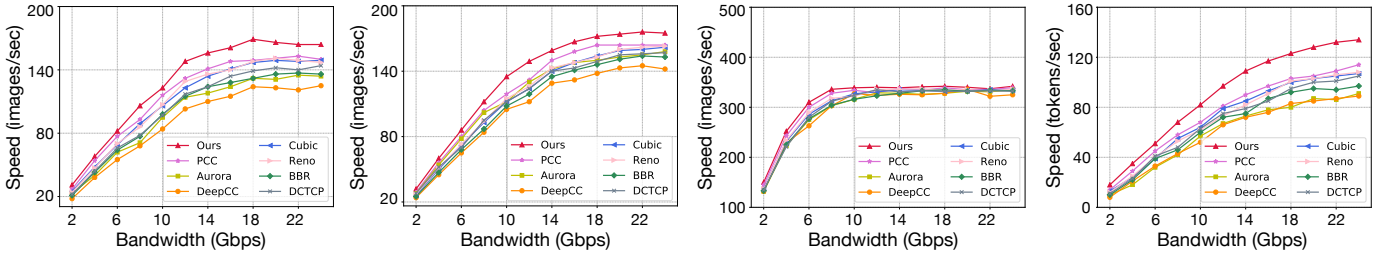
TABLE I: JCT statistics under different NIC bandwidth levels.

	25Gbps (mins)				20Gbps (mins)				15Gbps (mins)				10Gbps (mins)			
	avg.	**min.**	**max.**	**99th.**	**avg.**	**min.**	**max.**	**99th.**	**avg.**	**min.**	**max.**	**99th.**	**avg.**	**min.**	**max.**	**99th.**
Ours	3.96	1.12	6.23	6.10	4.41	1.28	8.09	7.92	5.54	1.74	10.29	9.20	7.05	2.24	13.20	12.89
PCC [39]	4.60	1.35	7.21	7.18	5.12	1.33	8.82	8.43	6.32	1.83	11.18	10.88	8.42	2.31	13.77	12.95
Aurora [40]	4.81	1.18	11.31	10.10	5.65	1.85	13.36	12.71	7.28	2.73	15.92	14.90	9.23	3.10	16.33	16.28
DeepCC [41]	4.94	1.61	12.83	10.42	5.55	2.09	13.22	13.01	7.16	2.65	14.96	14.24	9.62	2.73	18.31	15.25
CUBIC [36]	4.49	1.14	8.09	7.88	5.16	1.41	9.13	8.97	6.53	1.97	12.75	11.38	8.69	2.29	14.74	14.20
Reno [37]	4.53	1.26	7.24	7.03	5.19	1.52	9.93	9.82	6.47	2.01	13.01	11.89	8.72	2.31	14.82	14.80
BBR [32]	4.58	1.32	9.54	9.17	5.41	1.53	10.91	9.28	6.67	2.21	13.73	12.20	9.32	2.82	16.39	15.27
DCTCP [38]	4.55	1.46	8.38	8.26	5.24	1.59	9.72	9.37	6.72	2.16	13.23	11.30	9.12	2.41	15.93	15.73



(a) 25Gbps (b) 20Gbps (c) 15Gbps (d) 10Gbps

Fig. 8: Performance comparison under different NIC bandwidth levels.



(a) VGG19 (b) VGG16 (c) ResNet50 (d) GPT2

Fig. 9: Training speed at different bandwidth levels.

is more complex than the emulated scenarios that they focus on, e.g., Pantheon [42] emulating Internet paths. In a shared cluster, each sender only utilizes local measurements (i.e., RTT, loss) to generate a flow rate, without a mechanism to coordinate transmissions of concurrent flows.

Our schedulers receive link load information from switches, through which servers effectively cooperate to dynamically schedule outgoing flows for congestion avoidance. Different from centralized allocators [43] [44], we adopt a distributed framework and achieve dependency-aware bandwidth allocation. All these contribute to our training expedition.

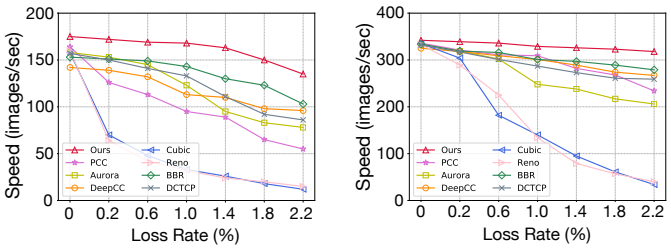
D. End-to-end Speed-up

To further examine training acceleration, we evaluate the training speeds of different DNN models under a range of bandwidth settings. Fig. 9 shows that our scheduler achieves higher training speeds than baseline control algorithms on all models. ResNet50 has respectively less speedup because it has fewer parameters (i.e., 98MB) and the communication time in a network with over 10Gbps bandwidth is much shorter than the computation time, leaving little room for improvement with flow scheduling. Other models are more communication intensive and benefit from our flow scheduling. Therefore, as the maximum bandwidth becomes larger, the improvement of the same model slightly decreases, e.g., VGG19 has a 22% speedup under 10Gbps bandwidth as compared to CUBIC, while achieving 14.2% speedup under 24Gbps bandwidth.

E. Tolerance to Random Factors

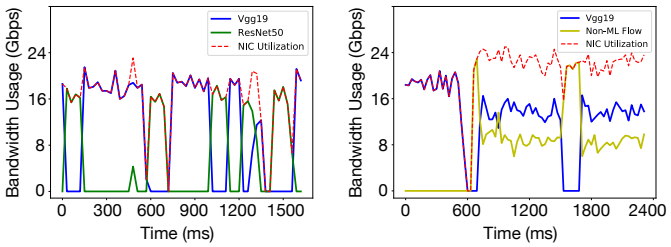
In a practical data center environment, besides traffic congestion, packet losses may happen due to random events such as physical fault of a cable or electromagnetic interference. To emulate such random factors, we use the tc tool to add a random packet loss rate on each link and evaluate the tolerance of our protocol.

We measure training speeds of VGG16 (communication-bound) and ResNet50 (computation-bound) under different packet loss rates. Fig. 10 shows that our protocol is highly resilient to packet losses. When the loss rate is larger than 1%, VGG-16 jobs only suffer from 4% performance degradation, and are 5.1× faster as compared to using CUBIC. We observe that the performance of TCP variants degrades very quickly, e.g., with only a 0.2% loss rate, training of VGG16 slows down by 62% with CUBIC and Reno. This is because the TCP sending window decreases exponentially whenever a packet loss is detected. Our scheduler achieves congestion-avoiding flow control, and has no hardwired mapping to packet-level events. Since our protocol is compatible with TCP, whenever packet losses occur, the scheduler uses TCP's re-transmission mechanism to re-transmit lost packets but does not reduce the sending rates computed.



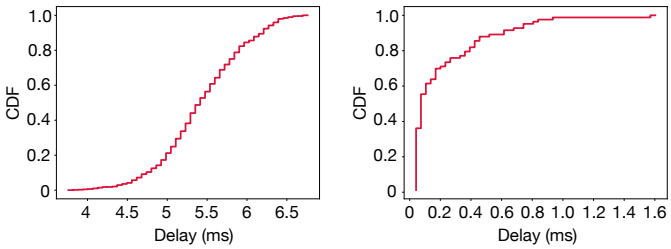
(a) VGG16 (b) ResNet50

Fig. 10: Training speed comparison at different loss rates.



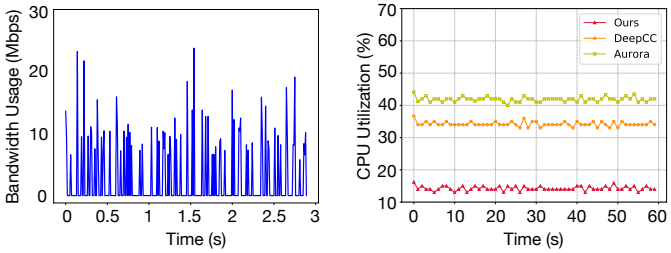
(a) Co-existing with ML Flow (b) Co-existing with non-ML Flow

Fig. 11: Bandwidth usage of competing flows.



(a) Negotiation Delay (b) Computation Delay

Fig. 12: Delay overhead of our schedulers at runtime.



(a) Bandwidth Load (a) Computation Load

Fig. 13: Load profiling of emulated switches and schedulers.

BBR and DCTCP do not rely on those packet events, making them more resilient to random factors while inducing more packet losses due to congestion at the same time [39]. For PCC, the rate decisions can be affected by such random noise: when a PCC flow is sent using a higher rate, a few packets are dropped due to the random factors while the flow rate has not reached the maximal available bandwidth, and PCC would lower its sending rate. A learning-based algorithm still needs a lot of online tuning, after its model is trained offline.

F. Flow behavior

We further look into competing flows in the same server. We run two all-reduce jobs training VGG19 and ResNet50, respectively. Each has a worker in the same server with 25Gbps NIC. Both have outgoing flows from the server to workers located on another server. As compared to Fig. 1), Fig. 11(a) shows that with our schedulers, flows are staggered to avoid potential congestion. This further brings benefits that are well-suited to the iterative pattern of ML training. As different ML jobs may have varying iteration times, staggering concurrent flows allows for the overlap of communication and computations between different jobs, which helps improves overall system efficiency.

To inspect co-existence of ML flows with non-ML flows, we run one VGG19 job with two workers, each located on a dedicated server. During training, we use iperf to inject one TCP flow between the two servers, at 600ms as shown in Fig. 11(b). Our scheduler estimates the available bandwidth by profiling the maximum sending rate over past 4 time slots and restricts further rate increment, hence achieving friendliness with non-ML flows controlled by TCP.

G. Scheduling Overhead

Each scheduler needs to exchange information about the current transmission schedule and link load estimation with emulated switches at the beginning of each time slot. Specifically, the scheduler first sends a $fetch\_feedback()$ request to the emulated switches and then fetches the feedback through the RPC framework. Then each scheduler employs Alg. 2 to compute the flow rate applied in the current time slot and adjust the schedule of remaining tensors. Fig. 12(a) reports the cumulative distribution function (CDF) of negotiation delays between schedulers and switches, which are measured as the duration between the start of $fetch\_feedback()$ request and the time when the link load estimations are obtained. We see that in 90% of the time slots, the scheduler can collect information within 6.2ms. In Fig. 12(b), we collect the computation times of schedulers to make rate decisions. Since the proposed algorithm can run in polynomial time and the rate adjustment is done in a heuristic manner, the time consumption is negligible. We further demonstrate the efficiency in simulation with thousands of jobs (Sec. VII).

In our framework, each emulated switch process emulates the negotiation behavior of one real switch on a dedicated server. It is scalable with the increase of the number of attached servers, due to three reasons. First, the messages exchanged between schedulers and switches are lightweight, making the bandwidth load on each emulated switch quite small. In our testbed, where one switch manages six servers, we measure the bandwidth load of the emulated switch over a period of time (including 50 time slots), as shown in Fig. 13(a). The bandwidth consumed by the negotiation traffic is below 10Mbps at most times, far less than the device capacity. Second, the computation and storage requirements are low. Each emulated switch only performs accumulation operations and the size of the load table grows linearly with the number of attached servers. On the other hand, modern programmable switches typically have memory capacities in the range of gigabytes. Third, in the machine learning cluster, GPU servers are typically inter-connected by hierarchical switches [9]. The Top-of-Rack (TOR) switch usually connects a group of GPUs first, then the aggregation switches in the upper layer. We are able to use multiple emulated switches to emulate the network

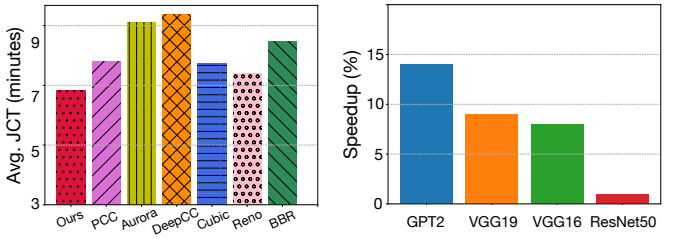


Fig. 14: Performance comparison with baselines and training speedup for different models.

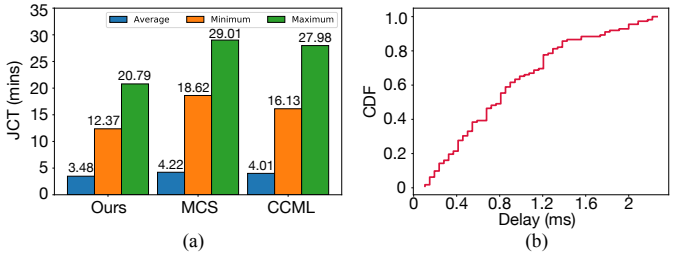


Fig. 15: Performance results with trace-driven simulations.

topology, thus avoiding the occurrence of a hotspot.

On the host side, we further compare the CPU load of our schedulers with that of learning-based algorithms implemented in the user space. We choose one scheduler and measure the average CPU utilization in 1-second intervals, while running the DNN training jobs described in Sec. VI-A. We do not track millisecond-level CPU load, since the high frequency monitoring would cause additional burden that affects the profiling results. Fig. 13(b) shows that our scheduler has significantly lower CPU consumption at runtime, i.e., 15% on average. DeepCC and Aurora incur much higher inference overhead, 35% and 42% of CPU utilization, respectively. With less CPU load, our scheduler incurs less performance interference with DNN training.

H. Large-scale Public Cloud Evaluation

To further verify the effectiveness of our designs in a larger scale, we conduct experiments on a 48-GPU AWS cluster. The cluster has 12 g4dn.12xlarge EC2 instances, each configured with 4 Nvidia T4 GPUs, 48 virtual CPU cores, 192 GB RAM, 50Gbps NIC and PCIe supporting a throughput around 15GB/s. Jobs are submitted following the configurations and arrival patterns as described in Sec. VI-A.

In a public cloud, the switch network connecting servers in the cluster is unknown to us, thus the entire network could be viewed as a logical switch. Instead of emulating switches as gRPC services, we maintain link load tables in the instances, and each instance estimates the traffic load on its outgoing links. This is reasonable in large-scale data centers, where the bandwidth bottleneck often lies at the server NICs [29]. Hence, scheduling bursty flows generated by concurrent jobs in the same server is essential for accelerating jobs training.

Fig. 14(a) shows the performance of our schedulers in terms of average JCT. Without emulating switch processes, our schedulers still achieve 8.5% improvement as compared to TCP variants. This is because our scheduler avoids the potential congestion on the host side, when co-located jobs compete for the network bandwidth. TCP variant protocols have very similar performance, i.e., Reno outperforms CUBIC with 4%. When allocating instances in AWS, EC2 by default places the instances in a way that spreads out all the instances across different regions to minimize correlated failures (called "Spread"). As compared to "Placement Group" strategy, which packs instances close together inside a region to achieve a low-latency network, the default instance placement leads to node-to-node communication traversing multiple switch hops [45]. This can introduce additional network variance and potential latency, which brings the advantage of Reno, because it uses more conservative rate adjustments as compared to CUBIC (more aggressive, and is likely to cause congestion when bursty traffic of multiple jobs occurs). The effect of unpredictable cross traffic also weakens the advantages of probing in BBR and makes learning-based methods low performance.

Fig. 14(b) shows the average speedup brought to different models, even in an environment with high-speed network environment (i.e., 50Gbps), model training still benefits from our scheduling protocol especially for large model like GPT2 (15% speed up). With the increasing size of ML models nowadays [2], our protocol can bring more potential performance improvement.

VII. Trace-driven Simulation

In this section, we compare our scheduler with SOTA coflow solutions. Since coflow frameworks with dependency consideration are not open-sourced [18] [14], we implement the core algorithm and conduct trace-drive simulations.

Simulator. Our simulator utilizes discrete-time simulation, i.e., we use *for-loop* to promote the progress of time slots, which is a common evaluation way in resource scheduling [46]. For each time slot, we compute/allocate the sending rates for uncompleted flows and calculate the transmitted volume of them by multiplying the sending rate and the time slot duration. A flow is completed when its accumulated transmit volume equals its flow size; otherwise, the duration will be accumulated as the flow's running time.

The arrival of DL workloads follows patterns extracted from Google traces, as mentioned in Sec. VI-A. The number of concurrent jobs can be more than one thousand. For each job, communication flows are generated following the iterative pattern: (1) within one iteration, the communication flows will start after its computation duration, which is calculated in advance for training one iteration with a full batch and divide it by the number of assigned workers. (2) when all flows in the current iteration are completed, the next iteration starts.

In our setting, each server in the simulation is equipped with 8 GPUs, and each GPU can be allocated to a worker or PS. The servers are interconnected following a hierarchical topology, where 10 servers form a rack connected to a ToR switch. Every 10 racks are further connected to an upper layer edge switch, and a total of 10 edge switches are linked to the aggregation switch in the top layer. In total, the simulation involves 1000 servers. Since our scheduler and coflow scheduling mechanisms are tolerant to packet-level events, we do not simulate low-level events such as packet

loss and re-transmissions. Instead, we define the bandwidth capacity of links in the different tiers as 10Gbps, 20Gbps, and 40Gbps, respectively. Flows that traverse the same link share the available bandwidth equally.

Baselines. We compare our scheduler to:
- MCS [17]: the representative coflow solution that considers scheduling multi-stage jobs with dependent coflows. MCS addresses the "start-after" dependency, which requires one coflow to start after the completion of the other flow. In the context of distributed training, that means in PS architecture, the PS needs to send broadcast flows after the completion of aggregation flows from workers; in AllReduce, one worker starts sending data to the subsequent worker after completely receiving data from the precedent worker. It solves a centralized problem and regards the whole cluster as a logical switch without considering the routing paths.

- CCML [47]: a congestion control method for ML jobs. It runs on the host side, profiles the iteration time of different jobs in advance, and aims to maximally interleave the communication phases of co-located jobs based on their iteration patterns.

A. Effectiveness

Fig. 15(a) shows the JCT statistics. Our scheduler outperforms MCS and CCML by 33.6% and 23.3% in terms of average JCT, respectively. The consideration of dependencies in the coflow problem is not suitable for pipelined transmission in distributed ML. When employed with MCS, the bandwidth from PSs to workers and the downstream bandwidth from one worker to its subsequent worker are underutilized, due to low pipeline degree in transmission. Similarly, the "Finish-before" dependency considered in coflows tends to cause two dependent coflows to finish at the same time [28]. However, this approach is not suitable for ML training and results in bandwidth waste and buffering of tensors at the PS node due to lower transmission rate from PS to worker. Our scheduler ensures that, at any given time slot during transmission, the accumulated size of transmitted data from worker to PS is nearly equal to the accumulated size of transmitted data from PS to worker (similar in case of AllReduce). By continuously monitoring dependency violations and dynamically adjusting the sending rate on the fly, we ensure the highest pipelining degree and maximal bandwidth utilization.

Furthermore, we incorporate load estimation techniques to effectively interleave concurrent bursty flows, achieving a similar objective as CCML. However, our approach not only takes into account congestion on the host side but also considers potential congestion along routing paths, with the help of in-network feedback from switches.

B. Efficiency

We present the CDF of the collected scheduling computation latency, as depicted in Fig. 15. The results indicate that 90% of the collected samples enable schedule decisions to be made within 1.5ms since the decision-making is done in a fully distributed manner. It further highlights the efficiency of our proposed algorithm, particularly when confronted with a large number of jobs.

VIII. Related Work

A. Networking in Distributed Training

Mai *et al.* [48] propose a communication layer that runs as a local process on workers and parameter servers. The layer uses each parameter server as a root and establishes a spanning tree to link all workers. Leaf workers send gradients to their parent workers, which aggregate the received gradients and push the results upstream toward the root. SwitchML [49] employs a programmable switch to aggregate gradients from multiple servers, update model parameters, and broadcast them to the workers, with the aim of reducing gradient traffic. ATP [4] extends in-network gradient aggregation to a multi-rack and multi-job cluster setting. Panama [2] equips the switch with an FPGA board to act as an in-network hardware accelerator, supporting floating-point gradient aggregation at line rate without compromising accuracy.

There have been efforts to prioritize flow scheduling in deep learning communication. Geryon [7] schedules CNN parameter transmissions at the network level by utilizing multiple flows with different priorities to transfer parameters of varying urgency levels. Large parameters are assigned to a higher urgency level, allowing them to initiate computation earlier. CEFS [50] employs a similar approach and extends the priority levels. Flows carrying data that can initiate computation earlier at a worker are assigned higher priorities. Flows directed toward workers with slower computation are also assigned higher priorities to alleviate the straggler problem.

Our flow control protocol is complementary to these existing efforts. Rather than altering the transmission order of tensors or the traffic volume delivered in the network, our schedulers precisely regulate the transmission rate of each flow to prevent congestion when multiple flows compete for bandwidth.

B. Data Center Flow Scheduling

Efficient bandwidth allocation is critical for achieving optimal application performance in a data center. pFabric [51] sets each flow with a separate priority for transmission. It controls each flow to start at line rate which falls back only upon packet loss events, in order to achieve the optimal flow completion time. NumFabric [52] enables an operator to specify how bandwidth is allocated amongst concurrent flows to minimize various objectives such as JCT and weighted fairness. Other solutions [53] [44] [43] usually collect the whole network status, and solve a centralized NUM problem to decide the optimal bandwidth allocation for each flow. We decompose the original NUM problem and develop a fully distributed scheduler, which optimizes the performance of distributed training workload.

Recently, receiver-driven techniques have also been proposed to proactively address bandwidth contention. For example, ExpressPass [54] manages congestion by controlling credit messages between switches and hosts, where the receiver sends credit messages, and the sender transmits a data message every time it receives a credit message. Homa [55] divides a flow into schedulable and non-schedulable parts; the non-schedulable packets carry flow states and acknowledge priorities for later packets. Rajasekaran *et al.* [47] achieved a

training speedup of two co-located DL jobs by tuning DCQCN for unfair bandwidth sharing.

In modern data centers, priority-based Flow Control (PFC) and ECN are widely employed for congestion control in an RDMA environment, but they do not support ML flow acceleration. Our prototype is implemented upon TCP stack, and our designs are fully compatible with RDMA techniques.

The flows generated in the communication phase of distributed training jobs are similar to coflows [56], where all flows in a collective group share a common finish time. Sincronia [57] employs a centralized coordinator and solves an ILP to decide the order of different coflows, with the objective of minimizing the makespan. Swallow [58] jointly considers coflow scheduling and compression in the shuffle stage of a MapReduce job. It leverages the idle CPU during the network transmission to compress data, and uses shortest-remaining-first algorithm to order contending coflows. Shafiee *et al.* [15] optimize the weight completion time of coflows, consider the whole data center as a logical switch and relax the original rate control problem to priority ordering of coflows. Tan *et al.* [16] consider the routing path of coflows and decide the rate for each coflow in an online manner. However, it neglects the dependency and solves the global optimization problem assuming the knowledge of all flows are well known to the controller. Tian *et al.* [18] study scheduling of coflows with dependency. It summarizes the potential dependency into two types, i.e., "start-after" and "finish-before", and controls the explicit flow rate in a centralized manner. Shafiee *et al.* [14] abstract each job as a directed acyclic graph (DAG) and the topology as a matrix (with sources/destinations as the rows/columns). It determines which packets of coflows should be sent at each time slot. However, it does not further extend the dependencies.

As have been stated in Echelonflow [28], existing coflow abstraction cannot handle the dependency in both data-parallel and pipeline-parallel ML jobs. We design a distributed framework to tune the flow rate dynamically, in order to cater to the unique dependencies.

IX. Conclusion

This paper proposes a fully distributed scheduler framework to expedite distributed training jobs and enhance overall throughput in a data center. On each server hosting DL tasks, a scheduler runs in the user space, interacts with the DL communication libraries, and exchanges information with switches to achieve congestion-avoiding flow rate control. We implement a prototype system with a software solution to emulate programmable switches. Experiments conducted on GPU clusters demonstrate that our schedulers can leverage the unique traffic patterns of distributed training and stagger transmissions of bursty ML flows to accelerate training. When compared to conventional TCP algorithms and state-of-the-art learning-based congestion control schemes, our approach achieves significant improvements in performance across various settings.

References

[1] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[2] N. Gebara, M. Ghobadi, and P. Costa, "In-network aggregation for shared machine learning clusters," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 829–844, 2021.

[3] H. Lim, D. G. Andersen, and M. Kaminsky, "3lc: Lightweight and effective traffic compression for distributed machine learning," *Proceedings of Machine Learning and Systems*, vol. 1, pp. 53–64, 2019.

[4] C. Lao, Y. Le, K. Mahajan, Y. Chen, W. Wu, A. Akella, and M. M. Swift, "Atp: In-network aggregation for multi-tenant learning." in *NSDI*, 2021, pp. 741–761.

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

[6] S. Shi, X. Chu, and B. Li, "Mg-wfbp: Merging gradients wisely for efficient communication in distributed deep learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 8, pp. 1903–1917, 2021.

[7] S. Wang, D. Li, and J. Geng, "Geryon: Accelerating distributed cnn training by network-level flow scheduling," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2020, pp. 1678–1687.

[8] Y. Bao, Y. Peng, Y. Chen, and C. Wu, "Preemptive all-reduce scheduling for expediting distributed dnn training," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2020, pp. 626–635.

[9] J. Li, H. Xu, Y. Zhu, Z. Liu, C. Guo, and C. Wang, "Lyra: Elastic scheduling for deep learning clusters," in *Proceedings of the Eighteenth European Conference on Computer Systems*, 2023, pp. 835–850.

[10] "Nccl," https://developer.nvidia.com/nccl, 2022.

[11] "ps-lite," https://github.com/dmlc/ps-lite, 2022.

[12] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: Meeting deadlines in datacenter networks," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 50–61, 2011.

[13] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "Pias: Practical information-agnostic flow scheduling for commodity data centers," *IEEE/ACM Transactions on Networking*, vol. 25, no. 4, pp. 1954–1967, 2017.

[14] M. Shafiee and J. Ghaderi, "Scheduling coflows with dependency graph," *IEEE/ACM Transactions on Networking*, vol. 30, no. 1, pp. 450–463, 2021.

[15] ——, "An improved bound for minimizing the total weighted completion time of coflows in datacenters," *IEEE/ACM Transactions on Networking*, vol. 26, no. 4, pp. 1674–1687, 2018.

[16] H. Tan, S. H.-C. Jiang, Y. Li, X.-Y. Li, C. Zhang, Z. Han, and F. C. M. Lau, "Joint online coflow routing and scheduling in data center networks," *IEEE/ACM Transactions on Networking*, vol. 27, no. 5, pp. 1771–1786, 2019.

[17] B. Tian, C. Tian, H. Dai, and B. Wang, "Scheduling coflows of multi-stage jobs to minimize the total weighted job completion time," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 864–872.

[18] ——, "Scheduling coflows of multi-stage jobs to minimize the total weighted job completion time," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 864–872.

[19] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.

[20] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[21] "Pytorch," https://github.com/pytorch/pytorch, 2022.

[22] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, "A unified architecture for accelerating distributed dnn training in heterogeneous gpu/cpu clusters," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 463–479.

[23] J. Romero, J. Yin, N. Laanait, B. Xie, M. T. Young, S. Treichler, V. Starchenko, A. Borisevich, A. Sergeev, and M. Matheson, "Accelerating collective communication in data parallel training across deep learning frameworks."

[24] "Gloo," https://github.com/facebookincubator/gloo, 2021.

[25] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine *et al.*, "Open mpi: Goals, concept, and design of a next generation mpi implementation," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 2004, pp. 97–104.

[26] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang, "Congestion control for large-scale rdma deployments," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 523–536, 2015.

[27] "Google-trace," https://github.com/google/cluster-data/, 2019.

[28] R. Pan, Y. Lei, J. Li, Z. Xie, B. Yuan, and Y. Xia, "Efficient flow scheduling in distributed deep learning training with echelon formation," in *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, 2022, pp. 93–100.

[29] C. DeCusatis, *Handbook of fiber optic data communication: a practical guide to optical networking*. Academic Press, 2013.

[30] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang, "Congestion control for large-scale rdma deployments," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 523–536, 2015.

[31] S. Low, "Optimization flow control with on-line measurement or multiple paths," in *Proceedings of the 16th International Teletraffic Congress*. Citeseer, 1999, pp. 237–249.

[32] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "Bbr: congestion-based congestion control," *Communications of the ACM*, vol. 60, no. 2, pp. 58–66, 2017.

[33] A. Narayan, F. Cangialosi, D. Raghavan, P. Goyal, S. Narayana, R. Mittal, M. Alizadeh, and H. Balakrishnan, "Restructuring endpoint congestion control," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 30–43.

[34] "Zeromq," https://github.com/zeromq/libzmq, 2021.

[35] "Transformers: State-of-the-art machine learning for pytorch, tensorflow, and jax." https://github.com/huggingface/datasets, 2021.

[36] S. Ha, I. Rhee, and L. Xu, "Cubic: a new tcp-friendly high-speed tcp variant," *ACM SIGOPS operating systems review*, vol. 42, no. 5, pp. 64–74, 2008.

[37] L. A. Grieco and S. Mascolo, "Performance evaluation and comparison of westwood+, new reno, and vegas tcp congestion control," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 2, pp. 25–38, 2004.

[38] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *Proceedings of the ACM SIGCOMM 2010 Conference*, 2010, pp. 63–74.

[39] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira, "Pcc: Re-architecting congestion control for consistent high performance," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015, pp. 395–408.

[40] N. Jay, N. Rotman, B. Godfrey, M. Schapira, and A. Tamar, "A deep reinforcement learning perspective on internet congestion control," in *International Conference on Machine Learning*. PMLR, 2019, pp. 3050–3059.

[41] L. Zhang, Y. Cui, M. Wang, K. Zhu, Y. Zhu, and Y. Jiang, "Deepcc: Bridging the gap between congestion control and applications via multi-objective optimization," *arXiv preprint arXiv:2107.08617*, 2021.

[42] F. Y. Yan, J. Ma, G. D. Hill, D. Raghavan, R. S. Wahby, P. Levis, and K. Winstein, "Pantheon: the training ground for internet congestion-control research," in *2018 USENIX Annual Technical Conference (ATC)*, 2018, pp. 731–743.

[43] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A centralized" zero-queue" datacenter network," in *Proceedings of the 2014 ACM conference on SIGCOMM*, 2014, pp. 307–318.

[44] J. Perry, H. Balakrishnan, and D. Shah, "Flowtune: Flowlet control for datacenter networks," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 421–435.

[45] "Ec2 placement group," https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/placement-groups.html, 2024.

[46] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proceedings of the 15th ACM workshop on hot topics in networks*, 2016, pp. 50–56.

[47] S. Rajasekaran, M. Ghobadi, G. Kumar, and A. Akella, "Congestion control in machine learning clusters," in *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, 2022, pp. 235–242.

[48] L. Mai, C. Hong, and P. Costa, "Optimizing network performance in distributed machine learning," in *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, 2015.

[49] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports, and P. Richtarik, "Scaling distributed machine learning with in-network aggregation," in *18th Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021, pp. 785–808.

[50] S. Wang, D. Li, J. Zhang, and W. Lin, "Cefs: compute-efficient flow scheduling for iterative synchronous applications," in *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, 2020, pp. 136–148.

[51] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pfabric: Minimal near-optimal datacenter transport," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 435–446, 2013.

[52] K. Nagaraj, D. Bharadia, H. Mao, S. Chinchali, M. Alizadeh, and S. Katti, "Numfabric: Fast and flexible bandwidth allocation in datacenters," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 188–201.

[53] B. C. Vattikonda, G. Porter, A. Vahdat, and A. C. Snoeren, "Practical tdma for datacenter ethernet," in *Proceedings of the 7th ACM european conference on Computer Systems*, 2012, pp. 225–238.

[54] I. Cho, K. Jang, and D. Han, "Credit-scheduled delay-bounded congestion control for datacenters," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017, pp. 239–252.

[55] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout, "Homa: A receiver-driven low-latency transport protocol using network priorities," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 221–235.

[56] A. Jajoo, Y. C. Hu, and X. Lin, "A case for sampling based learning techniques in coflow scheduling," *IEEE/ACM Transactions on Networking*, 2022.

[57] S. Agarwal, S. Rajakrishnan, A. Narayan, R. Agarwal, D. Shmoys, and A. Vahdat, "Sincronia: Near-optimal network design for coflows," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 16–29.

[58] Q. Zhou, P. Li, K. Wang, D. Zeng, S. Guo, and M. Guo, "Swallow: Joint online scheduling and coflow compression in datacenter networks," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 505–514.