Efficient Flow Scheduling in Distributed Deep Learning Training with Echelon Formation

Rui Pan[*]
Princeton University

Yiming Lei[*]
Max Planck Institute for Informatics

Jialong Li
Max Planck Institute for Informatics

Zhiqiang Xie
Stanford University

Binhang Yuan
ETH Zürich

Yiting Xia
Max Planck Institute for Informatics

ABSTRACT

This paper discusses why flow scheduling does not apply to distributed deep learning training and presents *EchelonFlow*, the first network abstraction to bridge the gap. EchelonFlow deviates from the common belief that semantically related flows should finish at the same time. We reached the key observation, after extensive workflow analysis of diverse training paradigms, that distributed training jobs observe strict computation patterns, which may consume data at different times. We devise a generic method to model the drastically different computation patterns across training paradigms, and formulate EchelonFlow to regulate flow finish times accordingly. Case studies of mainstream training paradigms under EchelonFlow demonstrate the expressiveness of the abstraction, and our system sketch suggests the feasibility of an EchelonFlow scheduling system.

CCS CONCEPTS

• **Networks** → **Cloud computing**; **Data center networks**;

KEYWORDS

Flow Scheduling, Data Center Networks, Deep Learning

ACM Reference Format:
Rui Pan, Yiming Lei, Jialong Li, Zhiqiang Xie, Binhang Yuan, and Yiting Xia. 2022. Efficient Flow Scheduling in Distributed Deep Learning Training with Echelon Formation. In *The 21st ACM Workshop on Hot Topics in Networks (HotNets '22), November 14–15, 2022, Austin, TX, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3563766.3564096

[*]Rui and Yiming contributed equally. This work was done during Rui and Zhiqiang's internship at MPI-INF and MPI-SWS, respectively.

HotNets '22, November 14–15, 2022, Austin, TX, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9899-2/22/11.
https://doi.org/10.1145/3563766.3564096

Training paradigm	CoFlow compliance	EchelonFlow arrangement
DP - AllReduce	✓	Same flow finish time
DP - PS	✓	Same flow finish time
PP	×	Staggered flow finish time
TP	✓	Same flow finish time
FSDP	×	Staggered Coflow finish time

Table 1: Mainstream distributed deep learning training paradigms in multi-tenant GPU clusters, including Data Parallelism (DP), with AllReduce and Parameter Server (PS) architectures, Pipeline Parallelism (PP), Tensor Parallelism (TP), and Fully-Sharded Data Parallelism (FSDP).

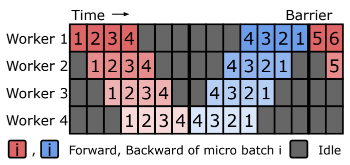
1 INTRODUCTION

Recent years have witnessed the rapid development of deep learning: each leap in the model quality comes with increased scales of neural networks, from AlexNet [30] with 61M parameters in 2012 to MT-NLG [50] with 530B parameters in 2022. Various parallel strategies (Table 1) have been adopted by distributed deep learning training (DDLT) frameworks [16, 26, 33, 39, 44, 49] to accommodate the ever-growing model sizes. As a result, communication among distributed workers, especially over a shared, highly dynamic network with competing training jobs, has become a notable bottleneck of the training process [33, 62].

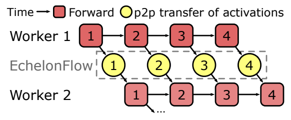
The networking community has a long history of resolving bandwidth contentions with flow scheduling, from individual flow scheduling [8, 9, 20, 58] to Coflow scheduling [13, 14, 23, 34, 60]. Surprisingly, despite the popularity of DDLT applications, we have found no flow scheduling solution supporting the diverse DDLT paradigms (Table 1) in GPU clusters! Our analysis suggests two reasons.

The *first reason* is due to the challenge of defining *a global optimization goal* across training jobs. As will be introduced in §2, the various DDLT paradigms implement drastically different workflows, which may translate into incompatible network requirements causing network-wide optimization to diverge. As such, communication optimizations for DDLT [10, 11, 19, 24, 27, 31, 36, 37, 43, 46, 54, 56, 61, 63] focus on data parallelism only, and most work conduct per-job optimization, with estimations of the available bandwidth.

Pioneering explorations for flow scheduling in DDLT [27, 37], also limited to data parallelism, faced exactly this problem. Particularly, CadentFlow [27] identified multiple performance metrics, e.g., weights, deadlines, and priorities,



(a) Computation timeline. Communications are omitted for simplicity.



(b) Computation-communication dependencies between two consecutive workers across micro-batches.

Figure 1: Workflow of Pipeline Parallelism (exemplified with GPipe [21]).

which may pull the optimization from different directions; MLNet [37] proposed to schedule flows by priorities, but how to set priorities to reflect application needs is unknown.

The *second reason* is the lack of *network abstraction* for DDLT. The Coflow [12] network abstraction for traditional cluster applications falls short in DDLT. Coflow defines a collection of semantically-related flows and minimizes the completion time of the last flow[1]. This goal motivates the optimizer to schedule the flows to finish at the same time. Oftentimes in DDLT, though, the followup computations consuming the flow data do not start at the same time.

Taking pipeline parallelism (Fig. 1) for example, each worker computes on sequential micro-batches and sends the results to a successor worker. Consecutive workers have data dependencies, and the computations per worker follow the order of input data. For high training throughput, GPU workers must be well coordinated to preserve the pipeline throughout the training lifetime. Delay or reordering of data may increase GPU idleness (the grey areas) and reduce training efficiency. To match this strict computation pattern, data flows across micro-batches should (ideally) finish in a staggered manner (Fig. 2c). Formulating the flows as a Coflow tends to finish them simultaneously (Fig. 2b), making the duration of this computation phase even longer than bandwidth fair sharing!

Through extensive workflow analysis, we generalize this observation to other DDLT paradigms: regardless of the great diversity, *each DDLT paradigm has a unique, pre-defined computation pattern that regulates the finish times of flow transmissions.* These computation patterns, which are essentially computation dependencies (i.e., DAG) and times, are prevalent in distributed applications. Yet, the repetitiveness of DDLT jobs, e.g., similar or identical computations across training layers and iterations, makes it possible to extract the patterns through computation profiling and convey the application-level guidelines to network flows.

Following this insight, we aspire to fill the gap of flow scheduling in DDLT. We propose the *EchelonFlow network abstraction* to finish flows according to strict DDLT computation patterns, and with EchelonFlow comes our *global optimization goal* of minimizing communication time while maintaining the computation patterns, like preserving the arrangement of an echelon formation [1]. *EchelonFlow is the first network abstraction for flow scheduling in diverse DDLT paradigms*. It is also extensible to future DDLT paradigms, as long as their computation patterns can be profiled.

Contributions of the paper are as follows.

- We formally define EchelonFlow and formulate a global optimization goal for it (§ 3.2).
- We prove important properties of EchelonFlow. Particularly, EchelonFlow scheduling can minimize completion times of mainstream DDLT paradigms, and EchelonFlow is a superset of Coflow (§ 3.3).
- Through case studies, we show the expressiveness of EchelonFlow by presenting popular DDLT paradigms with the EchelonFlow abstraction (§ 4).
- We sketch the system implementation to discuss the practicality of EchelonFlow scheduling (§ 5).

2 BACKGROUND

2.1 Distributed Deep Learning Training

Various parallel strategies have been proposed to accelerate DDLT. Here, we briefly review mainstream training paradigms (Table 1), with a focus on their communication patterns.

Data Parallelism (DP) is the most basic parallel strategy, where, as shown in Fig. 4a, each worker maintains a complete copy of the model and conducts forward and backward propagations on its local mini-batch of training data. The gradients generated by each worker are synchronized per training iteration, and popular schemes for gradient exchange communications include *parameter server* [26, 32] and MPI-style collective operators such as *AllReduce* [16, 33, 47].

The *parameter server* (PS) architecture (Fig. 4b) contains a logical PS node and a set of worker nodes. During training, each worker fetches the model from the PS, runs forward and backward computations, and pushes the gradients to the PS; while the PS aggregates the gradients from all the workers and updates the parameters. In *AllReduce*, nodes pass gradients to their neighbors along a ring by running the `all-reduce` collective communication operator, which further splits into a `reduce-scatter` followed by an `all-gather` operation. For an m-worker ring, each operation has $m-1$ steps, each containing $m-1$ data transfers.

Pipeline Parallelism (PP) [21, 40–42] is a form of model parallelism to distribute the model across workers. In PP, the model is partitioned into multiple stages each including consecutive layers of the neural network, and each stage is assigned to a worker. To maximize the system throughput, each training mini-batch is further split into micro-batches, so that computations can be executed as a pipeline. PP demands point-to-point communications for activations and their gradients to be exchanged between consecutive stages.

[1]Another optimization goal is to ensure all flows in a Coflow meet a common deadline. Flows missing the deadline can be aborted. This goal does not apply to DDLT, since data loss generally hurts model convergence.

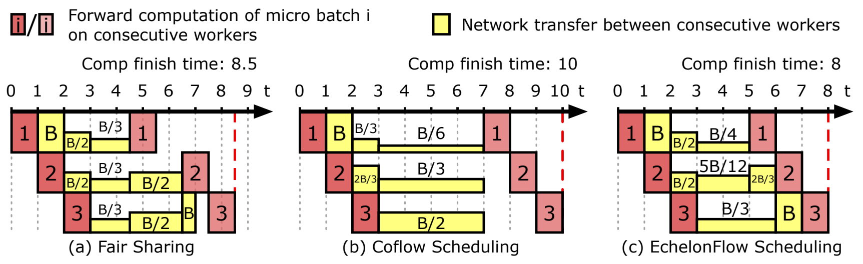


Figure 2: Motivating example for Pipeline Parallelism (Fig. 1): forward computations on three micro-batch; the predecessor worker sends activations of size $2B$ to the successor worker over a B-bandwidth link. EchelonFlow scheduling is optimal, with staggered flow finish times matching the computation pattern. Coflow makes all flows finish simultaneously and is worse than naive bandwidth fair sharing.

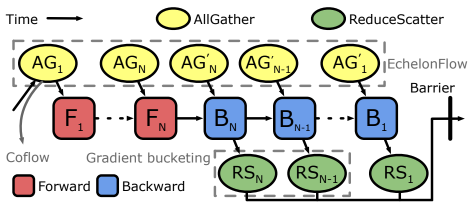
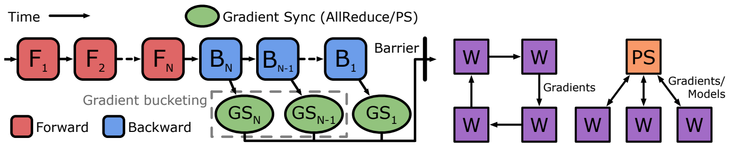


Figure 3: Workflow of Fully-Sharded Data Parallelism for one training iteration on one worker. Round-cornered squares denote computations, and ellipses denote communications.



(a) Workflow of Data Parallelism for one training iteration on one worker.
(b) AllReduce and Parameter Server architecture.

Figure 4: Overview of Data Parallelism. (a) Workflow and (b) implementations.

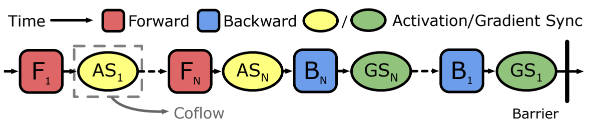


Figure 5: Workflow of Tensor Parallelism for one training iteration on one worker.

In the GPipe [21] realization (Fig. 1), each node sends the output activations to the successor node in the forward phase and sends the gradients of its input activations to its predecessor node in the backward phase. Later PP implementations [40–42] follow the same principle. They create similar computation pipelines, while reordering computations and data transmissions based on the data dependency to reduce the computation idleness (e.g., the grey areas in Fig. 1a).

Tensor Parallelism (TP) is another form of model parallelism introduced by Megatron [49] (Fig. 5) for even larger models. In the forward phase, all nodes must run an `all-reduce` to aggregate the activations generated by the forward computation from the local shard of parameters assigned to each node; while in the backward phase, all nodes will run another `all-reduce` for the corresponding gradients.

Fully-Sharded Data Parallelism (FSDP) [3], invented by ZeRO [44, 45] (Fig. 3), is a modification of vanilla DP for model scaling, where the parameters are sharded among all nodes, and computation and communication are done layer-wise. Each node collects the sharded parameters for the current layer with `all-gather` before the forward/backward computation and discards them afterwards to make room for the next layer. After the backward computation, an additional `reduce-scatter` is executed to dispatch the gradient shards to the corresponding nodes for synchronization.

2.2 Coflow

Coflow [12] is a network abstraction for traditional cluster applications, e.g., MapReduce [15], Spark [59], Pregel [38], and Dryad [22]. It conveys application-level semantics by identifying a collection of flows sharing a common performance goal, e.g., minimizing the completion time of the latest flow[1]. This goal implies the set of flows should finish at the same time, consistent with a common scenario where a computation task can only start after all flows from the previous communication stage have finished. Yet, Fig. 2 suggests DDLT jobs may require flows to finish at different times, and thus requires a new network abstraction. We will explain in §4 that Coflow can only present a subset of DDLT paradigms (Table 1).

3 ECHELONFLOW

Next, we detail the EchelonFlow network abstraction: its intuition, formal problem formulation, and proofs of properties.

3.1 Intuition

The design of EchelonFlow draws inspiration from echelon formations [1]. Intuitively, computation units in DDLT mimic aircraft units in an echelon military formation. The *arrangement*, or relative unit positions, in the echelon formation maps to the DDLT computation pattern. In Fig. 6, the diagonally arranged aircraft map to pipeline parallelism in DDLT (the dashed parallelogram) where computation units are pipelined across micro-batches on the same worker (blocks 1-4 of the same color on each worker in Fig. 1a).

An echelon arrangement can be represented by its *shape* and the *distance* between the units (Fig. 6a). For a computation arrangement, the "shape" is predetermined by the training paradigm based on its specific workflow (as in §2), and the "distance" is the duration of each computation unit, which can be profiled by running a few training iterations. In EchelonFlow, we describe the computation arrangement with an *arrangement function*, which contains the "shape" and "distance" of the arrangement.

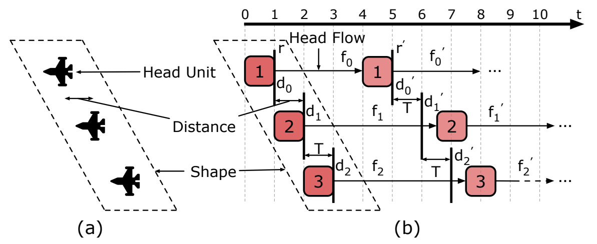


Figure 6: Intuition. (a) Aircraft arrangement of an echelon formation. (b) Computation arrangement of an EchelonFlow between two consecutive workers under Pipeline Parallelism (Fig. 1). Two EchelonFlows $H = \{f_0, f_1, f_2\}$ and $H' = \{f_0', f_1', f_2'\}$ in (b); r and r' denote the reference time of H and H'; d_0, d_1, d_2, d_0', d_1', d_2' denote the ideal finish time of f_0, f_1, f_2, f_0', f_1', f_2'.

Analogous to formation flying where aircraft units maintain the echelon arrangement, the computation arrangement in DDLT is determined by data transmissions that stall the computations. As Fig. 1b shows, the computation pipeline on a worker depends on data from the predecessor worker. Hence, the key to maintaining the computation arrangement in DDLT is to manage the finish times of the traffic flows. To speed up training, we should minimize the communication time while maintaining the computation arrangement.

To this end, we define *ideal finish time* as the earliest finish time of a flow in theory. This definition enables us to minimize the communication overhead in DDLT for the (unrealistically) ideal case of an infinitely fast network. Assuming zero data transmission time, the ideal flow finish time (d_0, d_1, d_2 in Fig. 6b) is its start time. If the computation unit generating the data flow misses the computation arrangement (due to delay of the previous flow), then the ideal flow finish time should be further advanced to offset the delay. In Fig. 6b, f_1' and f_2' start late because of delays of the previous flows f_1 and f_2. Their ideal finish times d_1' and d_2' are thus set to earlier than their start times, to give them opportunities to transmit faster and catch up with the computation arrangement.

How much should we offset the ideal finish time? Given the arrangement of an echelon formation (Fig. 6a), the location of each unit can be derived by "reference" to the location of the head unit. Likewise, in EchelonFlow, we define the *reference time* as the start time of the flow that starts first (*head flow*). The ideal finish time of the head flow is its start time and also the reference time. Ideal finish times of later flows can all be derived from the "reference time", one by one using the relative "distance" in the arrangement function. In this way, a DDLT job recalibrates the computation arrangement whenever a new EchelonFlow is generated.

3.2 Problem Formulation

*Definition 3.1 (**EchelonFlow**).* An EchelonFlow is a set of flows with related ideal finish times, where the relation is represented by an arrangement function of the reference time.

$H = \{f_0, f_1, ..., f_{|H|-1}\}$ is an EchelonFlow with reference time r, where $|H|$ is the cardinality (i.e., the number of flows) in H, and the flows follow the ascending order of the start time. The set $D = \{d_0, d_1, ..., d_{|H|-1}\}$ contains the ideal finish time d_j of each flow $f_j \in H$, and s_0 is the start time of f_0. Then $d_0 = r = s_0$, and we have an arrangement function $g(D, r)$.

*Definition 3.2 (**Flow Tardiness**).* The tardiness of a flow is its actual finish time exceeding its ideal finish time.

Let d be the ideal finish time of a flow f and e be the flow's actual finish time, the tardiness t_f of f is:

$$t_f = e - d \tag{1}$$

We define tardiness to differentiate from most flow scheduling work that minimizes flow completion time. Tardiness regulates flows regarding their ideal finish times, rather than their flow start times. This definition allows computation units to realign with the arrangement per EchelonFlow. If optimizing with flow completion time, after flows delay, later EchelonFlows cannot recover the arrangement.

*Definition 3.3 (**EchelonFlow Tardiness**).* The tardiness of an EchelonFlow is the maximum tardiness of all its flows.

For EchelonFlow H, following the above notations, let e_j be the actual finish time of flow f_j, corresponding to the ideal finish time d_j. The tardiness t_H of H is:

$$t_H = max(e_j - d_j), \quad 0 \le j < |H| \tag{2}$$

Following the intuition in § 3.1, the tardiness of all the flows in an EchelonFlow should remain the same if the EchelonFlow constantly maintains the computation arrangement. The definition of maximum tardiness helps to reduce the difference in tardiness among individual flows.

Optimization Objective

Naturally, based on our earlier definitions, the optimization objective of EchelonFlow scheduling is tardiness minimization. For an individual EchelonFlow H, particularly, the objective is to minimize its tardiness t_H:

$$\textbf{Minimize:} \quad z = t_H \tag{3}$$

For multiple EchelonFlows, the objective is to minimize the sum of their tardiness. For a set EchelonFlows $\mathcal{H} = \{H_0, H_1, ..., H_{|\mathcal{H}|-1}\}$, where $|\mathcal{H}|$ is the cardinality and t_{H_i} is the tardiness of EchelonFlow $H_i \in \mathcal{H}$, the objective is:

$$\textbf{Minimize:} \quad \hat{z} = \sum_{0 \le i < |\mathcal{H}|} t_{H_i} \tag{4}$$

The objective can be easily adjusted to the weighted sum of individual EchelonFlows' tardiness, should there be a proper way to assign weights to different DDLT jobs.

3.3 Properties

Here we list important properties of EchelonFlow and give a high-level overview of the proofs. We have proved these properties formally in a technical report [2].

Property 1: EchelonFlow scheduling minimizes completion times of popular DDLT paradigms.

Tardiness minimization aims to advance computation units while maintaining the desirable computation arrangement, which ultimately speeds up training. We prove it case-by-case for the popular DDLT paradigms in Table 1 [2].

Property 2: EchelonFlow is a superset of Coflow.

Coflow can be presented as a special EchelonFlow where all the flows share a common ideal finish time (Eq. 5). In this case, by definition, the tardiness of every flow is its finish time minus the start time of the first flow. Our EchelonFlow optimization objective of minimizing the maximum tardiness among all the flows (Eq. 3) becomes minimizing Coflow completion time. This property makes EchelonFlow compatible with traditional cluster applications covered by Coflow.

Property 3: EchelonFlow scheduling is NP-hard.

Coflow scheduling is NP-hard [14], so the superset problem EchelonFlow scheduling is also NP-hard.

Property 4: Coflow scheduling algorithms can be adapted to EchelonFlow scheduling at the same complexity.

There exists a one-to-one mapping between EchelonFlow and Coflow metrics. In this sense, we can adapt Coflow scheduling algorithms to EchelonFlow scheduling, with a different metric for evaluating flows. In MADD [14], for example, in intra-EchelonFlow scheduling, we estimate the latest flow that has the largest tardiness, rather than the longest flow completion time as for Coflow; in inter-EchelonFlow scheduling, we rank EchelonFlows by each EchelonFlow's tardiness (Eq. 2), instead of the Coflow completion time. This mapping does not change the algorithm complexity.

4 CASE STUDIES

Mainstream DDLT paradigms in Table 1 can all be described by EchelonFlow. We demonstrate the expressiveness of EchelonFlow by showing each paradigm's arrangement function.

Case I: Coflow-Compliant Paradigms

As discussed in § 3.3, Coflow is a special EchelonFlow whose flows share the same ideal finish time. For a Coflow in the form of an EchelonFlow H, the ideal finish time d_j of flow $f_j \in H$ follows the arrangement function below, where r is the reference time, i.e., the start time of the first flow f_0.

$$d_j = r, \quad 0 \leq j < |H| \tag{5}$$

Next, we show how to group flows into Coflows for different Coflow-compliant DDLT paradigms.

Data Parallelism. The workers exchange gradients after backward computations (Fig. 4a), and training frameworks bucket gradients of several layers to perform `all-reduce` across workers [33]. In the ***AllReduce*** architecture (Fig. 4b), these gradient transmissions form a Coflow, because the training can move on to the next bucket only after they all finish. In ***parameter server*** (Fig. 4b), besides these Coflows for gradient synchronizations from workers to the PS, on the reverse path, the PS updates the model and sends the weights to all workers. These flows form another Coflow, as the completion of them all signifies the start of the next training iteration.

Tensor Parallelism. Megatron [49] (Fig. 5) contains two types of `all-reduce` operations. The first is for activation synchronization from all the workers in each layer of forward computation, and the second is for gradient synchronization per layer in the backward pass. The all-to-all flows in each `all-reduce` fall into a Coflow, as they altogether barrier computation in the next layer.

Case II: Pipeline Parallelism

The GPipe [28] (Fig. 1b) pipelines computations for successive micro-batches of data on each worker, so the data flows these computations depend on should preserve staggered finish times following the computation pipeline. As Fig. 1b shows, the flows from one worker to another form an EchelonFlow. The arrangement function of such an EchelonFlow H is as below, with notations similar to Eq. 5.

$$d_j = \begin{cases} r, & j = 0 \\ d_{j-1} + T, & 1 \leqslant j < |H| \end{cases} \tag{6}$$

The ideal finish time d_0 of the first flow f_0 is its start time, or the EchelonFlow's reference time r. The ideal finish time d_j of each flow f_j afterwards is time T later than that of the previous flow, where T is the computation time on one micro-batch of data. As explained in §3.1, T can be obtained from computation profiling on the training framework.

Other PP variations [40–42] form EchelonFlows similarly, while reordering computations for different micro-batches. As long as the data depencies are determined, relations between the data flows can also be expressed as an arrangment function, albeit more complicated than Eq. 6.

Case III: Fully-Sharded Data Parallelism

ZeRO/FSDP [3, 44] (Fig. 3) uses the `all-gather` collective primitive to gather weights from all nodes before each layer's forward and backward computations. Flows in each `all-gather` collective form a Coflow. These Coflows along the computation timeline further form an EchelonFlow, following a pipeline-like pattern as in GPipe (Fig. 1b). The arrangement function is thus similar to Eq. 6.

$$d_{c_i} = \begin{cases} r_{c_0}, & i = 0 \\ d_{c_{i-1}} + T_{fwd}, & 1 \leqslant i \leqslant n-1 \\ d_{c_{i-1}} + T_{bwd}, & n \leqslant i \leqslant 2n-1 \end{cases} \tag{7}$$

For an n-layer neural network, let C_i be the i_{th} Coflow, then $C_0 - C_{n-1}$ and $C_n - C_{2n-1}$ belong to the forward and backward phase, respectively. The reference time r_{c_0} is the reference time of the first Coflow C_0, which is the start time of its first flow. Since all flows in a Coflow share the same ideal finish time (Eq. 5), the (single) ideal finish time of each Coflow d_{c_i} is time T_{fwd} or T_{bwd} later than the previous Coflow $d_{c_{i-1}}$ depending on whether it lies in the forward or backward phase. T_{fwd} and T_{bwd} can both be profiled.

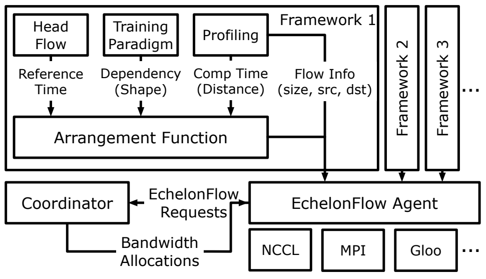


Figure 7: EchelonFlow scheduling system diagram.

The `reduce-scatter` operations after the backward computation of each layer, from the communication's perspective, are equivalent to gradient synchronizations in DP (Fig. 4b). Similarly, the `reduce-scatter` flows in each gradient bucket across different workers are in a Coflow.

5 SYSTEM SKETCH

EchelonFlow targets at DDLT in GPU clusters, where training jobs share the network bandwidth and GPUs can be fragmented [25, 56]. Because EchelonFlow relies on accurate profiling of the computation time to construct the arrangement function, it is best suited for the mainstream training configuration with dedicated, monolithic GPUs. As performance isolation in GPU sharing advances [6, 52], EchelonFlow may apply to GPU-shared training in the future.

In this section, we get insights from DDLT communication scheduling and Coflow scheduling systems to sketch our system design. Fig. 7 is our envisioned system diagram.

We are inspired by ByteScheduler [43] to build an *EchelonFlow Agent* as a shim layer between DDLT frameworks and message-passing backends, e.g., NCCL [5], MPI [55], and Gloo [4]. It collects EchelonFlow information across DDLT frameworks and implements coordinated flow scheduling decisions by issuing communication calls to the backends. For each training instance, the framework breaks down the workflow into EchelonFlows, like in § 4, based on the training paradigm used. For each EchelonFlow, it reports the arrangement function and per-flow information (the size, source, and destination) to the agent via a library of EchelonFlow APIs.

The agent sends the EchelonFlow requests to a *Coordinator* for EchelonFlow scheduling. As discussed in § 3.3, we expect the coordinator to run a heuristic algorithm adapted from Coflow scheduling, e.g., MADD [14]. Such algorithms would rerun per EchelonFlow arrival/departure or per scheduling interval. We propose to improve the scalability by revising them to maintain the scheduling decision throughout the DDLT lifetime leveraging the iterative nature of DDLT jobs.

We follow the common practice to enforce the schedules through flow priorities [13, 23, 34]. The agent stores flow data into priority queues based on their allocated bandwidth, and calls message-passing backends through weighted sharing of network bandwidth among the queues. This approach, from the communication call's perspective, is essentially the same as gradient reordering in many DDLT communication scheduling systems [19, 24, 43]. Yet, unlike their strategy of single-job optimization, our EchelonFlow agent supports communication scheduling across DDLT jobs.

6 RELATED WORK

EchelonFlow is a direct improvement to DDLT communication scheduling solutions [10, 11, 19, 24, 27, 31, 36, 37, 43, 46, 54, 56, 61, 63]. Compared to their limited scope of DP, EchelonFlow provides a comprehensive abstraction for diverse training paradigms. By actively shaping the network dynamics across jobs, EchelonFlow is also more effective than most prior work's passive reactions to available network bandwidth per job [10, 11, 19, 24, 37, 56, 61, 63].

EchelonFlow is motivated by Coflow [7, 13, 14, 23, 34, 60, 64], but deviates from Coflow's assumption that all flows in a collective group have a common finish time. The arrangement function of EchelonFlow captures general computation dependencies, giving EchelonFlow richer expressiveness (e.g., PP and FSDP) than Coflow. EchelonFlow incorporates inter-Coflow dependencies in the design, e.g., concatenating Coflows in FSDP (§4), similar to inter-Coflow scheduling in multi-stage applications with DAGs [35, 48, 51, 53, 57].

EchelonFlow is orthogonal to communication-aware task scheduling [17, 18, 29]. EchelonFlow takes the computation DAG as input to construct the arrangement function and coordinates jobs holistically through flow scheduling, whereas their approach is for each job to consider the network condition while building the DAG. We solve the common problem of accelerating job completion from different angles.

7 CLOSING REMARKS

Our proposal of *EchelonFlow* provides a breakthrough to flow scheduling for DDLT. Core to the proposal lie the *new network abstraction* to unify different workflows across training paradigms under the generic *arrangement function*, and the global optimization goal of *tardiness minimization* to enforce different flow finish times. This novel solution opens up two new avenues for network-for-machine-learning research. First, it shows potential to regulate the network proactively, e.g., via EchelonFlow scheduling, to coordinate machine learning applications, rather than the current practice of per-application optimization under what the network grants. Second, it urges the networking community to integrate network management systems and machine learning systems, like the EchelonFlow scheduling system, to make network a first-class resource for machine learning applications.

ACKNOWLEDGMENTS

We thank Hong Zhang, Florin Dinu, and Wei Bai for early feedback of the work. We are also grateful to the anonymous reviewers for their insightful reviews.

REFERENCES

[1] 2022. Echelon Formation. https://en.wikipedia.org/wiki/Echelon_formation.

[2] 2022. EchelonFlow technical report. https://anonymous.4open.science/r/EchelonFlow_report.

[3] 2022. FairScale. https://github.com/facebookresearch/fairscale.

[4] 2022. Gloo. https://github.com/facebookincubator/gloo.

[5] 2022. NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/nccl.

[6] 2022. NVIDIA Multi-Instance GPU User Guide. https://docs.nvidia.com/datacenter/tesla/pdf/NVIDIA_MIG_User_Guide.pdf.

[7] Saksham Agarwal, Shijin Rajakrishnan, Akshay Narayan, Rachit Agarwal, David Shmoys, and Amin Vahdat. 2018. Sincronia: Near-optimal network design for coflows. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 16–29.

[8] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pfabric: Minimal near-optimal datacenter transport. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 435–446.

[9] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. 2017. PIAS: Practical information-agnostic flow scheduling for commodity data centers. *IEEE/ACM Transactions on Networking* 25, 4 (2017), 1954–1967.

[10] Yixin Bao, Yanghua Peng, Yangrui Chen, and Chuan Wu. 2020. Preemptive all-reduce scheduling for expediting distributed dnn training. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 626–635.

[11] Minsik Cho, Ulrich Finkler, David Kung, and Hillery Hunter. 2019. Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. *Proceedings of Machine Learning and Systems* 1 (2019), 241–251.

[12] Mosharaf Chowdhury and Ion Stoica. 2012. Coflow: A networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*. 31–36.

[13] Mosharaf Chowdhury and Ion Stoica. 2015. Efficient coflow scheduling without prior knowledge. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 393–406.

[14] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. 2014. Efficient coflow scheduling with varys. In *Proceedings of the 2014 ACM conference on SIGCOMM*. 443–454.

[15] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[16] Shaoduo Gan, Jiawei Jiang, Binhang Yuan, Ce Zhang, Xiangru Lian, Rui Wang, Jianbin Chang, Chengjun Liu, Hongmei Shi, Shengzhuo Zhang, et al. 2021. Bagua: scaling up distributed learning with system relaxations. *Proceedings of the VLDB Endowment* 15, 4 (2021), 804–813.

[17] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. 2016. Altruistic Scheduling in Multi-Resource Clusters. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 65–80.

[18] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. 2016. GRAPHENE: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 81–97.

[19] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy Campbell. 2019. Tictac: Accelerating distributed deep learning with communication scheduling. *Proceedings of Machine Learning and Systems* 1 (2019), 418–430.

[20] Chi-Yao Hong, Matthew Caesar, and P Brighten Godfrey. 2012. Finishing flows quickly with preemptive scheduling. *ACM SIGCOMM Computer Communication Review* 42, 4 (2012), 127–138.

[21] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems* 32 (2019).

[22] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. 59–72.

[23] Akshay Jajoo, Y Charlie Hu, and Xiaojun Lin. 2019. Your coflow has many flows: sampling them for fun and speed. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 833–848.

[24] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. 2019. Priority-based parameter propagation for distributed DNN training. *Proceedings of Machine Learning and Systems* 1 (2019), 132–145.

[25] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 947–960.

[26] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. 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)*. 463–479.

[27] Sangeetha Abdu Jyothi, Sayed Hadi Hashemi, Roy Campbell, and Brighten Godfrey. 2020. Towards An Application Objective-Aware Network Interface. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*.

[28] Chiheon Kim, Heungsub Lee, Myungryong Jeong, Woonhyuk Baek, Boogeon Yoon, Ildoo Kim, Sungbin Lim, and Sungwoong Kim. 2020. torchgpipe: On-the-fly pipeline parallelism for training giant models. *arXiv preprint arXiv:2004.09910* (2020).

[29] Dzmitry Kliazovich, Johnatan E Pecero, Andrei Tchernykh, Pascal Bouvry, Samee U Khan, and Albert Y Zomaya. 2016. CA-DAG: Modeling communication-aware applications for scheduling in cloud computing. *Journal of Grid Computing* 14, 1 (2016), 23–39.

[30] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25 (2012), 1097–1105.

[31] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. 2021. ATP: In-network Aggregation for Multi-tenant Learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 741–761.

[32] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 583–598.

[33] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. 2020. PyTorch distributed: experiences on accelerating data parallel training. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3005–3018.

[34] Shuhao Liu, Li Chen, and Baochun Li. 2018. Siphon: Expediting Inter-Datacenter Coflows in Wide-Area Data Analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 507–518.

[35] Yang Liu, Wenxin Li, Keqiu Li, Heng Qi, Xiaoyi Tao, and Sheng Chen. 2016. Scheduling dependent coflows with guaranteed job completion

time. In *2016 IEEE Trustcom/BigDataSE/ISPA*. IEEE, 2109–2115.

[36] Liang Luo, Peter West, Jacob Nelson, Arvind Krishnamurthy, and Luis Ceze. 2020. Plink: Discovering and exploiting locality for accelerated distributed training on the public cloud. *Proceedings of Machine Learning and Systems* 2 (2020), 82–97.

[37] Luo Mai, Chuntao Hong, and Paolo Costa. 2015. Optimizing network performance in distributed machine learning. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*.

[38] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.

[39] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 561–577.

[40] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 1–15.

[41] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. 2021. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning*. PMLR, 7937–7947.

[42] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. Efficient large-scale language model training on GPU clusters using megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[43] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 16–29.

[44] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[45] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3505–3506.

[46] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 785–808.

[47] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[48] Mehrnoosh Shafiee and Javad Ghaderi. 2021. Scheduling Coflows With Dependency Graph. *IEEE/ACM Transactions on Networking* 30, 1 (2021), 450–463.

[49] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[50] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, et al. 2022. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv preprint arXiv:2201.11990* (2022).

[51] Penghao Sun, Zehua Guo, Junchao Wang, Junfei Li, Julong Lan, and Yuxiang Hu. 2021. Deepweave: Accelerating job completion time with deep reinforcement learning-based coflow scheduling. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*. 3314–3320.

[52] Cheng Tan, Zhichao Li, Jian Zhang, Yu Cao, Sikai Qi, Zherui Liu, Yibo Zhu, and Chuanxiong Guo. 2021. Serving DNN models with multi-instance gpus: A case of the reconfigurable machine scheduling problem. *arXiv preprint arXiv:2109.11067* (2021).

[53] Bingchuan Tian, Chen Tian, Haipeng Dai, and Bingquan Wang. 2018. Scheduling coflows of multi-stage jobs to minimize the total weighted job completion time. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 864–872.

[54] Raajay Viswanathan, Arjun Balasubramanian, and Aditya Akella. 2020. Network-accelerated distributed machine learning for multi-tenant settings. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 447–461.

[55] David W Walker and Jack J Dongarra. 1996. MPI: a standard message passing interface. *Supercomputer* 12 (1996), 56–68.

[56] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica. 2020. Blink: Fast and generic collectives for distributed ml. *Proceedings of Machine Learning and Systems* 2 (2020), 172–186.

[57] Junchao Wang, Huan Zhou, Yang Hu, Cees De Laat, and Zhiming Zhao. 2017. Deadline-aware coflow scheduling in a dag. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 341–346.

[58] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. 2011. Better never than late: Meeting deadlines in datacenter networks. *ACM SIGCOMM Computer Communication Review* 41, 4 (2011), 50–61.

[59] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A Fault-Tolerant abstraction for In-Memory cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 15–28.

[60] Hong Zhang, Li Chen, Bairen Yi, Kai Chen, Mosharaf Chowdhury, and Yanhui Geng. 2016. Coda: Toward automatically identifying and scheduling coflows in the dark. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 160–173.

[61] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. 2017. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 181–193.

[62] Zhen Zhang, Chaokun Chang, Haibin Lin, Yida Wang, Raman Arora, and Xin Jin. 2020. Is network the bottleneck of distributed training?. In *Proceedings of the Workshop on Network Meets AI & ML*. 8–13.

[63] Zhenwei Zhang, Qiang Qi, Ruitao Shang, Li Chen, and Fei Xu. 2021. Prophet: Speeding up Distributed DNN Training with Predictable Communication Scheduling. In *50th International Conference on Parallel Processing*. 1–11.

[64] Yangming Zhao, Kai Chen, Wei Bai, Minlan Yu, Chen Tian, Yanhui Geng, Yiming Zhang, Dan Li, and Sheng Wang. 2015. Rapier: Integrating routing and scheduling for coflow-aware data center networks. In *2015 IEEE Conference on Computer Communications (INFOCOM)*. IEEE, 424–432.