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

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 contentsions 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,
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.
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 layerwise. 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.
whenever a new EchelonFlow is generated.

Way, a DDLT job recalibrates the computation arrangement
the communication overhead in DDLT for the (unrealistically)
time of a flow in theory. This definition enables us to minimize
3.2 Problem Formulation

represented by an arrangement function of the reference time.

Analogous to formation flying where aircraft units main-
the echelon arrangement, the computation arrangement
in DDLLT 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 DDLLT 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 DDLLT for the (unrealistically)
ideal case of an infinitely fast network. Assuming zero data
transmission time, the ideal flow finish time \(d_0, d_1, d_2\)
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 \(f_0, f_1, f_2, f'_0, f'_1, f'_2\).

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 \(f_0, f_1, f_2, f'_0, f'_1, f'_2\).

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 \(f_0, f_1, f_2, f'_0, f'_1, f'_2\).

\[ \text{Property 1: EchelonFlow scheduling minimizes completion time of popular DDLT paradigms.} \]

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 \(g(D, r)\).

\[ \text{Definition 3.2 (Flow Tardiness). The tardiness of a flow is} \]
\[ \text{its actual finish time exceeding its ideal finish time.} \]

Let \(d\) be the ideal finish time of a flow \(f\) and \(e\) be the flow’s
actual finish time, the tardiness \(\hat{t}_f\) of \(f\) is:

\[ \hat{t}_f = e - d \quad \text{(1)} \]

We define tardiness to differentiate from most flow sched-
uling 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.

\[ \text{Definition 3.3 (EchelonFlow Tardiness). The tardiness of an} \]
\[ \text{EchelonFlow is the maximum tardiness of all its flows.} \]

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 \(\hat{t}_H\) of \(H\) is:

\[ \hat{t}_H = \max(e_j - d_j), \quad 0 \leq j < |H| \quad \text{(2)} \]

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

\[ \text{Optimization Objective} \]

Naturally, based on our earlier definitions, the optimization
objective of EchelonFlow scheduling is tardiness mini-
mization. For an individual EchelonFlow \(H\), particularly, the
objective is to minimize its tardiness \(\hat{t}_H\):

\[ \text{Minimize: } \hat{z} = \hat{t}_H \quad \text{(3)} \]

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

\[ \text{Minimize: } \hat{z} = \sum_{0 \leq i < |\mathcal{H}|} t_{H_i} \quad \text{(4)} \]

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].

\[ \text{Property 1: EchelonFlow scheduling minimizes completion} \]
\[ \text{times of popular DDLT paradigms.} \]
EchelonFlow

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 $f_0$.

$$d_j = r, \quad 0 \leq j < |H|$$

(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.

$$d_j = r, \quad j = 0$$

$$d_j = d_{j-1} + T, \quad 1 \leq j < |H|$$

(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 dependencies are determined, relations between the data flows can also be expressed as an arrangement 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.

$$d_i = \begin{cases} r_{cs}, & i = 0 \\ d_{i-1} + T_{fwd}, & 1 \leq i \leq n-1 \\ d_{i-1} + T_{bwd}, & n \leq i \leq 2n-1 \end{cases}$$

(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 $r_{cs}$ 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 $d_{ci}$ is time $T_{fwd}$ or $T_{bwd}$ later than the previous Coflow $d_{ci-1}$ depending on whether it lies in the forward or backward phase. $T_{fwd}$ and $T_{bwd}$ can both be profiled.
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 Echelon-Flow 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

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


