Ampel: Scheduling at the Network Cut in ML Training

Valerio Torsiello
ETH Zurich
vtorsiello@ethz.ch

Ayush Mishra
ETH Zurich
aymishra@ethz.ch

Sushovan Das
ETH Zurich
susdas@ethz.ch

Lukas Röllin
ETH Zurich
roellinl@ethz.ch

Tommaso Bonato
ETH Zurich
tommaso.bonato@inf.ethz.ch

Torsten Hoefler
ETH Zurich
torsten.hoefler@inf.ethz.ch

Laurent Vanbever
ETH Zurich
lvanbever@ethz.ch

Abstract

The size and communication patterns of modern ML training workloads place significant strain on datacenter fabrics. When bandwidth demand exceeds capacity, flows experience slowdowns and iteration time grows larger. Fine-grained load-balancing such as packet spraying cannot fully resolve this issue, yet it can shift the bottleneck from individual links to groups of links partitioning the network. In this work, we present Ampel: a system to schedule ML training flows at the network cuts. It leverages packet-spraying's ability to spread traffic evenly across all available paths to simplify the view of the topology into one only containing potential bottlenecks, and then bridges this new simplified model with past work on coflow scheduling. Our simulated experiments show that Ampel can reduce average training iteration time by up to 18% compared to state-of-the-art ML schedulers.

CCS Concepts

• **Networks** → **Data center networks**; **Network design principles**.

Keywords

Datacenter Network, ML Training, Coflow Scheduling

ACM Reference Format:
Valerio Torsiello, Ayush Mishra, Sushovan Das, Lukas Röllin, Tommaso Bonato, Torsten Hoefler, and Laurent Vanbever. 2026. Ampel: Scheduling at the Network Cut in ML Training. In *Workshop on Networks for AI Computing (NAIC '26), August 17–21, 2026, Denver, CO, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3789240.3828736

1 Introduction

Large-scale distributed machine learning (ML) has emerged as a dominant workload in modern clusters, with training jobs spanning hundreds to thousands of GPUs . These workloads exhibit a distinct communication pattern: high-volume, synchronized, and with network-wide traffic bursts, where large collectives must complete before computation can proceed [14, 29, 32]. As a result, network performance directly determines overall training efficiency.

A central challenge in such environments is bandwidth contention under *overdemand*: when GPUs attempt to inject traffic at a higher rate than the network can handle. Overdemand arises naturally in ML clusters due to a combination of factors: reduced capacity stemming from intentional oversubscription in network cores [14, 29], failures, or link degradations [29]; and increased demands originating from job fragmentation and heavier and wider collectives (e.g., AllToAll(v)) [8, 22, 27].

Given that overdemand is common in practice, an important question is how to mitigate its impact. Recent works advocate scheduling approaches to reduce contention between ML jobs. Systems such as CASSINI [32] and Crux [9] stagger job execution or assign priorities and paths based on workload characteristics to improve flow completion times. While useful, these approaches share a fundamental limitation: *they rely on a link-centric model of contention, assuming bottlenecks arise at specific paths or links.* Yet, modern fabrics increasingly rely on fine-grained multipath load balancing, such as packet spraying [3, 5, 25], to spread the load across all paths.

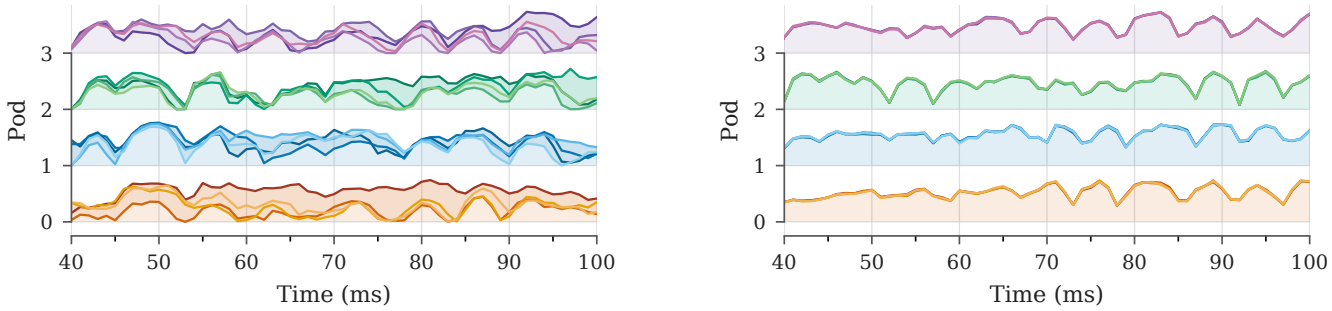
In this paper, we argue that scheduling for ML jobs must be rethought in the packet-spraying era. Our key insight is that packet spraying has fundamentally changed how contention manifests: *instead of being localized to individual links, bottlenecks arise at sets of links which partition the network.* Overdemand will therefore occur when the total demand crossing a *network cut* exceeds its capacity, making cuts the ideal abstraction to focus on. Via this insight, we reframe how a scheduling approach should act: instead of having to gauge the demand on every single link and the path taken by each flow, one must only consider which flows cross which oversubscribed network cuts and how to allocate bandwidth to them, which is remarkably simpler.

To formulate the scheduling problem, we observe the tight dependencies of ML computation on collective sets of flows, and thus build upon the coflow abstraction [11], which can capture this. We introduce the **Network Cut Coflow Scheduling** (NCCS) problem as a variation of classical coflow scheduling, and present **Ampel**, a dynamic scheduling system for ML workloads to tackle it in a realistic setting. It operates in epochs and performs four steps: (i) identifying oversubscribed network cuts, (ii) tracking active coflows and their demands, (iii) ranking coflows by applying coflow scheduling at the *network cuts* (rather than at the NICs), and (iv) scheduling coflows while ensuring that their aggregate demand fits within the available cut capacity.

We evaluate Ampel through simulations with realistic ML traces. By avoiding overfilling bottleneck cuts, Ampel reduces congestion, improves network utilization, and allows scheduling decisions to adapt dynamically over time. Our results show that Ampel improves

NAIC '26, Denver, CO, USA

ACM ISBN 979-8-4007-2467-1/26/08
https://doi.org/10.1145/3789240.3828736



(a) Single-Path Ultra-Ethernet **(b) Packet Spraying Ultra-Ethernet**

Figure 1: Packet trims over time by 16 aggregation switches (grouped by pod) in a Fat Tree. Under packet spraying, switches in the same pod display identical congestion patterns due to the even traffic spread. As a result, bottlenecks will not be localized to individual links, but to the set of links connecting partitions of the network.

job iteration time from 3% to 18% in oversubscribed scenarios compared to Crux, by better matching admitted demand to available cut capacity.

In summary, this paper makes the following contributions:

- We identify *network cuts* as the right abstraction for reasoning about overdemand in packet-spraying ML networks.
- We formulate ML scheduling as a network-cut-aware coflow scheduling problem.
- We design Ampel, a practical dynamic scheduler that schedules coflows based on cut capacity and enforces decisions using lightweight priorities.
- We evaluate Ampel on realistic traces, showing it improves iteration time in oversubscribed network conditions.

This work does not raise any ethical issues.

2 Motivation

This section characterizes bandwidth contention in ML clusters, its root causes, and how related work addresses them.

2.1 Bandwidth Contention in ML-Clusters

Compared to more traditional workloads, bandwidth contention for ML training traffic differs significantly in terms of its periodicity, intensity, and synchronicity [14, 29].

ML flows are high volume. Compared to traditional cloud computing, GPUs running ML workloads engage in fewer and less varied connections, but with higher bandwidth demands [14, 29]. Devices frequently transmit data at line rate, reaching speeds of up to 800 Gbps [5]. To sustain such throughput and minimize flow completion times, these deployments rely on fast RDMA-based transports such as Infiniband, RoCEv2 and Ultra-Ethernet (UET) [3, 16].

ML flows are synchronized. Large models often rely on multiple forms of parallelism—such as data, pipeline, tensor, and expert parallelism [18, 21, 28, 34, 37, 38]. These require frequent collective communication phases to exchange gradients and model states. Since these phases are typically synchronized across tens or hundreds of GPUs, many high-bandwidth flows are injected into the network simultaneously. This wide traffic surge places significant pressure on the fabric and can lead to severe bandwidth contention.

ML flows compete network-wide. Recent measurement studies [15] show that congestion in RDMA-based AI training fabrics is not only intense and highly synchronized, but also shifting deeper into the network. Congestion is no longer confined to network edge; instead, backpressure can propagate upstream and often manifests in the network core, fundamentally changing where and how contention emerges.

2.2 Understanding *Overdemand*

Congestion is the result of traffic being injected into network links at a higher rate than they can handle. Congestion Control tackles this problem by rate-limiting the flows to reduce buffer buildups and packet drops. Yet, it does not address the root cause of the issue, which we call *overdemand*: when the sum of the *desired* rates (mainly the line rates) of all flows crossing a link or set of links exceeds the available capacity.

Overdemand on individual links may be the result of poor load-balancing (LB): an asymmetric spread of flows can result in greater strain for certain paths—this is common with coarse-grained LB. Recent transports such as UET [3, 7, 20] and MRC [5] try to mitigate this by spraying packets across all available paths to maximize bandwidth utilization. *Packet Spraying* is a very fine-grained form of LB where paths are chosen per-packet, and it results in a very even spread of traffic. We observe this in the simulated experiments shown in Fig. 1, where the congestion patterns experienced by aggregation switches within the same pod are identical to each other when using packet spraying. This even spread is more effective; yet, no LB approach can neutralize overdemand when it arises at a *cut* of the network, i.e., the demand from the subgraph containing the traffic sources to that containing the destinations exceeds the capacity of the links connecting the two. Multiple factors can contribute to this issue, due to either a lowered capacity or an increased demand.

Reduced capacity. While the ideal practice for dedicated ML training is to build Full-Bisection Bandwidth networks [27], this can be

expensive at scale [40, 41]. As a result, some infrastructure providers opt for *oversubscription* [14, 29], generally placed closer to the network core to minimize its impact on congestion. However, with large job sizes, avoiding these oversubscribed regions entirely is challenging.

Additionally, *failures* and *link degradation* may also contribute to overdemand by reducing network capacity. For example, Alibaba reports 5K–60K link-flapping events per day introducing temporary performance degradation [29]. Such concerns motivate recent work on failure-aware transport, failover, and resilient multipath routing for datacenter networks [6, 7, 19]. The core issue is not asymmetry, which is handled by adaptive multipath mechanisms [7], but rather the transient overdemand that can ensue from the reduced capacity.

Increased demand. Collective Communication Libraries (CCLs) typically prioritize locality when constructing collectives, reducing hop count and latency. In principle, this approach should avoid the oversubscribed regions commonly found in the network core. Yet, in practice, locality can be difficult to achieve. The scale of modern ML workloads in multi-tenant clusters can result in job fragmentation [26, 30, 42]. Expert parallelism further exacerbates this, as its dynamic and fine-grained patterns make it even harder to mitigate contention through job placement or collocation alone [22].

2.3 Related Work

There are a number of strategies available to avoid and mitigate the effects of overdemand in large ML-training networks. As discussed earlier, traditional congestion control-based approaches *slow down* the senders involved in an episode of overdemand to minimize congestion and packet drops. This is not the most desirable outcome, as the slowdowns of these jobs in their communication phases can quickly add up and increase iteration time. Another available strategy is to produce these flows in a way that minimizes contention in the network via worker placement and deciding how these workers communicate. These considerations are often made by CCLs like [2, 10, 17, 23, 24, 35] . We consider these mechanisms orthogonal to our work—while they minimize overdemand, they are not able to completely eliminate it.

Recent works like [9, 32, 33] have observed that since ML flows are periodic and bursty, there is an opportunity to *stagger* the periods in which each participating flow injects traffic into the network. This is possible because not all participating flows need to complete at the same time in order for compute to make progress. This is especially true when the over-demand is created by competing jobs and different collectives that might not be dependent on each other—and therefore do not *have* to use the network at the same time.

CASSINI [32] staggers the start time of jobs in order to minimize the likelihood that they will create overdemand. It does so by profiling the ML jobs, and then computing their respective offsets to minimize the overlap in their communication phases. This strategy relies on ML jobs being predictable. However, this is not always true. As we show in Fig. 2, periods of contention can change how an ML job uses the network, limiting the efficacy of a job offset-based approach for minimizing traffic synchronicity.

An alternate strategy is utilizing priorities instead of fixed offsets. Priorities are more dynamic in nature (they can be changed on the fly) and can effectively stagger the times at which periodic traffic is injected into the network. Intuitively, this works by giving strict priority to some flows over others, thus allowing only a subset of the flows to use the link capacity at a given time. Once the higher priority flows have completed, the lower priority flows can utilize the network capacity. Systems like Crux [9] utilize such mechanisms and use the GPU intensity of a job to determine its priority. However, as we will show in §5, these static priorities are not always optimal and can create starvation for some jobs. Finally, both Crux and CASSINI are designed to work over a single path, while modern load balancing approaches like packet spraying use multiple paths and create overdemand over entire network cuts rather than single links.

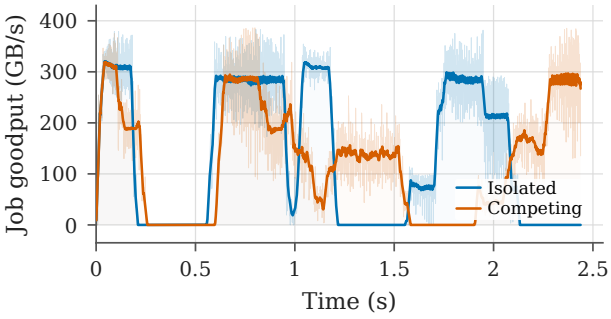


Figure 2: A training job's communication patterns may be affected when sharing the network with other jobs, hurting assumptions based on periodicity.

3 Scheduling at the Cut

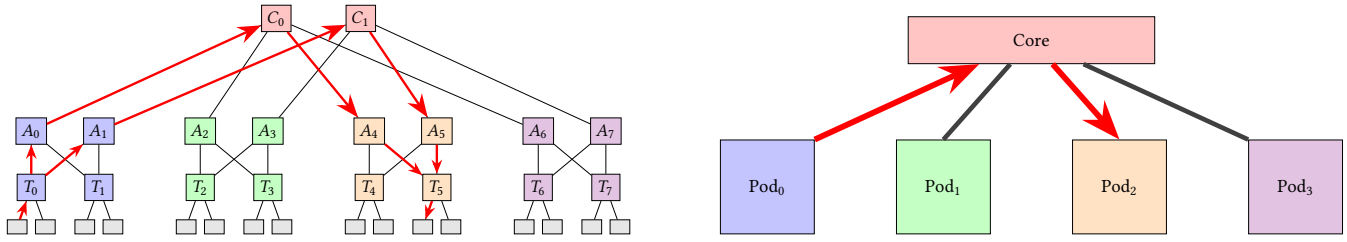
In this section, we present a new model for scheduling to address the shortcomings of other approaches. We highlight properties of fine-grained load-balancing/multi-pathing techniques that enable us to shift our focus to oversubscribed network cuts, and introduce a new abstraction to capture this behavior (§3.1). We then describe how existing flow scheduling techniques map to our model, and select coflow scheduling as the closest formulation to our problem (§3.2).

3.1 Focusing on the Bottlenecks

We present a model that simplifies our view of the network into one that only captures potential bottlenecks.

Packet spraying spreads flows over cuts. Our core insight lies in packet spraying's ability to utilize all available paths to a destination simultaneously: for example, in a 3-layer Fat-Tree (Fig. 3a), a flow being sent from one pod to another will utilize *all* uplinks from source pod to the core, and *all* downlinks from the core to destination pod. Therefore, the available outgoing (or incoming) capacity for all flows leaving (or entering) a certain network region is equal to the capacity of the cut partitioning it—in the example, if we partition a pod from the rest of the topology, the available capacity is that of two top-layer links.

Bottlenecks arise at oversubscribed cuts. Due to this property of packet spraying, when the graph is partitioned into two subgraphs (one with sources and one with destinations), no individual link



(a) 3-layer Fat Tree with oversubscription at the core.

(b) Abstracted view.

Figure 3: The even traffic spread given by packet spraying simplifies our view of the network from (a) into (b).

can become a bottleneck if the edge cut between them has multiple links (Fig. 1b), as long as all links in the cut can be used by all flows. Thus, instead of links, we compute demand over entire cuts. If a network cut is not oversubscribed, it will not see overdemand (with ideal packet spraying). Hence, to identify potential bottlenecks, *we only need to look at oversubscribed network cuts.*

The number of relevant cuts can be reduced. In this work, we focus on Fat Tree topologies with an oversubscribed core layer (and discuss more in §6). Such networks provide an intuitive way to reduce the number of relevant cuts, due to their hierarchical structure: we must only select, for each pod, the set of links connecting it to the core. This effectively makes each set of uplinks equivalent to a single big link, each pod equivalent to a single big node, and the core routers equivalent to a single big switch, as shown in Fig. 3b. Incidentally, this abstraction resembles the big-switch model often seen in scheduling literature.

The abstraction simplifies scheduling. This view of the topology results in a remarkable simplification of the scheduling problem: instead of tracking the usage of each individual link in the entire network and the path taken by each flow, our optimization lies simply in tracking the demand from and to each pod. Consequently, the scheduling algorithm is absolved from having to make routing decisions or trying to approximate a multi-commodity flow problem, and only needs to choose which flows are admitted on each big link.

3.2 Defining the Scheduling Problem

We have simplified the setting that scheduling will take place in, and now seek the most appropriate approach to schedule ML flows. Our high-level objective is to minimize the time it takes for GPUs to get back to compute work, instead of idly waiting for the communication phase to complete.

ML needs bulk synchronization. In selecting the branch of scheduling theory that most closely approximates our problem, our key insight was the need of ML models to perform bulk synchronization. This need is well captured by the coflow abstraction [11]: a coflow is defined as a set of flows sharing the same downstream dependency, which can't resume until all flows in the coflow have terminated; hence, the collective performance of all the flows in the coflow is more important than that of the individual flows.

Coflow scheduling is dynamic and flexible. Coflow scheduling has been extensively studied [4, 12, 13] where the objective is to minimize the average Coflow Completion Time (CCT, defined as the span of time from the start of its earliest flow to the end of its latest one) or to enforce completion within a certain deadline. Unlike the works on ML scheduling described in §2.3, coflow schedulers tend to be highly dynamic and are capable of reacting to the current flow demands. Additionally, determining what constitutes a coflow is also quite flexible; for example, in ML training, a coflow could contain just the flows within a given collective, but also all the flows involved in a job's entire training iteration.

We perform coflow scheduling over network cuts. These properties make coflow scheduling an appealing theoretical framework for our problem, which we dub the Network Cut Coflow Scheduling (NCCS) problem. We formally describe it in Appendix A.1 and prove its NP-hardness.

Coflow scheduling has the benefit of having a wealth of literature trying to approximate its optimal solution. We therefore choose it as a starting point for our algorithm: we can make use of *existing coflow scheduling techniques*, but rather than using the coflows queued at each NIC/host as their input (like prior work [4, 12, 13, 39]), we instead use the coflows queued at each oversubscribed cut. This constitutes step 3 in Ampel's algorithm (§4.1).

However, note that subtle but important differences separate us from a pure application of a coflow scheduler. Namely, the capacity of a network cut can be much greater than each NIC's line rate, and we can schedule multiple flows on each one. This is conveyed by step 4 in Ampel's algorithm (§4.1).

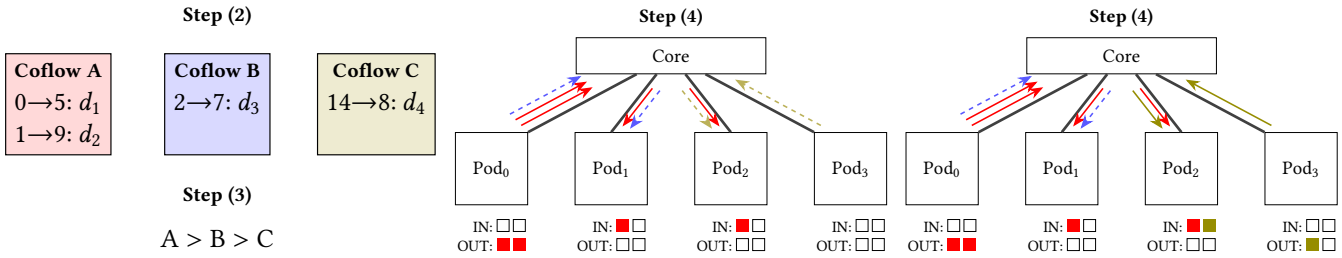
4 Ampel

We design Ampel to tackle NCCS in a realistic setting. First, we outline an algorithm to bridge the gap between coflow scheduling and our new abstraction (§4.1). Then, we discuss how to integrate this in a real network (§4.2).

4.1 Algorithm

We propose a centralized algorithm to tackle the NCCS problem. In Fig. 4 shows an example of a Fat Tree of 4 pods with 4 GPUs each, and OS 2:1 at the core. It proceeds in 4 steps:

(1) Identify all oversubscribed network cuts. When computing cut capacity, the controller must be aware of the network's default



(a) Identify and rank coflows. **(b) Schedule coflow A.** **(c) Skip B and schedule C.**

Figure 4: Overview of Ampel's scheduling algorithm.

structure and, if possible, of failures and link degradation. In the example, we take 4 cuts: the 2 links connecting each pod to the core.

(2) Identify all active coflows containing flows crossing any oversubscribed cuts, and the amount of data each device wants to send to each other device as part of that coflow. In Fig. 4a, we have 3 coflows. For example, coflow B wants GPU 2 (in pod 0) to send d_3 bytes to GPU 7 (in pod 1).

(3) Execute a coflow scheduling algorithm that ranks the coflows, using the network cuts rather than the NICs as the ports in the big switch model. This step is open-ended: the algorithm may be something as simple as "least bytes remaining" as well as something complex such as Varys [13] or Sincronia [4]. The operator may further customize it depending on the metric they want to optimize (e.g. fairness, GPU intensity, etc.).

(4) Iterate coflows in the order given by step 3 and greedily allocate bandwidth on the cuts to them based on their individual (src, dst) pairs. The amount of data to send is ignored: we only consider the bandwidth demand, i.e., the line rate. If all cuts still have capacity to support all pairs in the current coflow, schedule that coflow and reduce the cuts' capacity accordingly; else, do not schedule the coflow. In the example, the demands of unscheduled coflows are shown with dashed lines. Coflow B is skipped due to insufficient outgoing capacity from Pod 0.

Note that we may also include the incoming/outgoing links at the NICs as oversubscribed cuts. This can be useful if a single NIC is involved in multiple coflows: for example, if we group flows into coflows by their collective ID, and individual GPUs are performing multiple collectives simultaneously.

4.2 System Design

As in prior work on coflow scheduling, Ampel's decisions are made centrally by a controller, which runs the algorithm every "epoch". We assume each host knows which coflow (e.g. a global coflow ID) each flow belongs to, and that it can estimate how much data it will send towards each destination as part of that coflow. We do *not* assume that any host (nor the controller) can map a given coflow ID to all its constituent flows, as many of these might not have started yet.

Hosts must inform the controller each time a new coflow starts, so that the scheduling algorithm may include it during the next epoch. They should also update the controller on the remaining data to be sent for each coflow. Once the scheduling step is completed, the controller tells the hosts whether each of their coflows has been scheduled. The scheduling is enforced via network priorities. Only two priority queues are necessary for this: one for flows within scheduled coflows, and another one (of lower priority) for all other flows.

5 Evaluation

We evaluate Ampel against a baseline without scheduling and Crux, a state-of-the-art ML scheduler. Our preliminary evaluation demonstrates that network-cut-aware scheduling can indeed improve ML job performance under *overdemand*.

Simulation setup. We run our experiments on htsim [31], a packet-level simulator for datacenter networks, and use real LLM traces collected via ATLAHS [36], which records computation and communication operations, including their dependencies. We simulate a network that uses packet spraying, as defined by the Ultra-Ethernet stack [3], with trimming enabled and sender-driven NSCC. Load balancing uses Mixed-REPS, combining the REPS [7] and bitmap [20] approaches. The topology is a 3-layer Fat-Tree with 8 pods, 128 nodes, and 200 Gbps links. Depending on the experiment, each switch port uses either two or three priority queues: the top-priority queue is reserved for trimmed ("header-only") packets, while 1 or 2 additional other queues are for data traffic.

Different approaches. The baseline has no prioritization, hence requiring *one* additional priority queue. For Ampel and Crux, the buffer capacity is split into *two* additional priority queues, keeping the total capacity constant. In Ampel, these correspond to scheduled (medium-priority queue) and unscheduled (low-priority queue) traffic. Ampel's coflow granularity is set to job-level (i.e., flows with the same job-id are one coflow). Ampel's controller epochs are set to 5 ms; it uses Sincronia's (BSSI) algorithm [4] to rank coflows (step 3 in §4.1). Crux is implemented without path selection, as we use packet spraying; its priority compression step assumes flows to collide if they traverse the same network cut.

Workloads. In this experiment, we consider 8 Llama7B jobs running for two iterations. Each job is assigned 16 GPUs and uses a ZeRO-1 parallelism strategy. Each iteration consists of 98 `AllGather` and 98 `ReduceScatter` collectives of varying tensor sizes. Each collective is shaped as 4 parallel inter-host rings: one between all GPUs

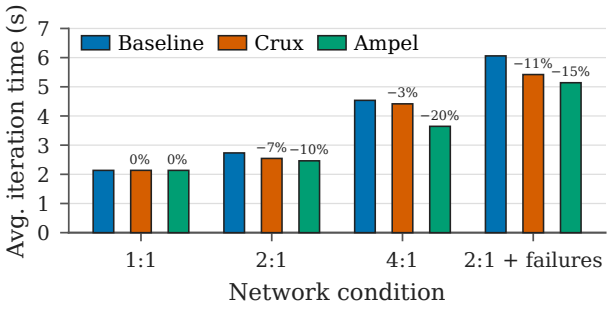


Figure 5: Scheduling improves iteration time under overdemand. Ampel consistently outperforms Crux.

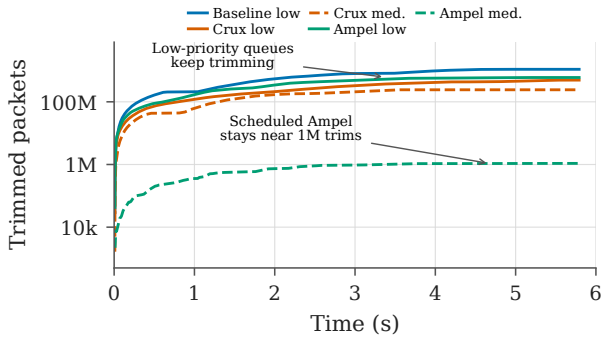


Figure 6: Trims at different priority queues. Scheduled flows in Ampel do not face overdemand, significantly reducing trimmed packets in medium-priority queues.

in rank 0 (i.e., one per host, four in total), another between all GPUs in rank 1, and so on.

The traces were collected from real AI workloads run on the Alps supercomputing cluster [1], where each host contained 4 GPUs and 4 NICs (one per GPU). In our simulation setup, each compute device represents 1 GPU/NIC, with GPUs belonging to the same host being placed under the same ToR. To capture job fragmentation (§2.2), we randomize the host placement while preserving intra-host locality.

Preliminary results. Fig. 5 displays the average completion time of the 8 jobs for all three approaches given various degrees of oversubscription at the core layer, as well as a scenario with 18 core layer link failures. Under no oversubscription (1:1), all approaches perform similarly as no overdemand exists, hence no benefit in prioritization. By introducing oversubscription at the top layer, the links connecting each pod to the core become bottlenecks. As oversubscription increases, Ampel consistently achieves the lowest iteration time, ranging from 3% to 18% lower than Crux.

6 Discussion & Future Work

Both Crux and Ampel outperform the baseline by reducing flow overlap at bottlenecks, improving effective bandwidth utilization (§2.3). We examine the sources of Ampel's gains, highlighting key design trade-offs and future directions.

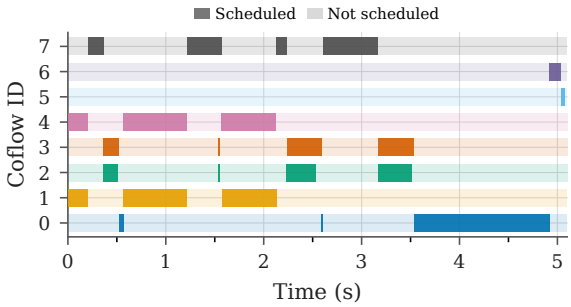


Figure 7: Ampel allows dynamic prioritization. The set of scheduled coflows (jobs) changes over time.

Focusing on bottlenecks. Ampel's gains stem from accurately estimating the network cut capacity and selecting coflows without overfilling bottlenecks (step 4 in §4.1). As a result, scheduled coflows do not induce overdemand, avoiding contention under limited bandwidth. This is reflected in Fig. 6: the number of trimmed packets over time at 4:1 oversubscription (no packets were dropped in any of the experiments, only trimmed). The curves (log-scale) correspond to trims at different priority queues across the network. Ampel's medium-priority queues (i.e., flows in scheduled coflows) have orders of magnitude fewer trimmed packets, indicating that these flows experience little to no congestion. Future work can explore how load-balancing techniques other than packet spraying impact this behavior, and if Ampel's core insights still apply to coarser multipathing.

Dynamic scheduling. Ampel can update the set of scheduled coflows every epoch, adapting to demand. Fig. 7 demonstrates the evolution of scheduled coflows (jobs), showcasing Ampel's ability to perform dynamic prioritization. However, this flexibility may not always impact performance significantly. For example, in this scenario, less frequent updates yielded similar iteration times. Looking forward, dynamic scheduling can enable richer policies such as starvation avoidance and fairness across coflows.

Scheduling granularity. Ampel supports multiple scheduling granularities through different coflow definitions. Beyond job-level coflows (shown so far), we also evaluated finer-grained alternatives such as grouping by collective. In our experiments, coarser granularities (e.g., job-level) performed better, while collective-level coflows (likely too fine-grained) offered no improvement. Exploring intermediate or adaptive granularities is a promising direction; for instance, silence periods between communication bursts could help separate coflows, albeit requiring fine-grained profiling.

Different topologies. Our evaluation focuses on Fat-Tree topologies with core oversubscription, but an important question is how Ampel generalizes to other designs. For example, asymmetry within pods (e.g., due to failures) would introduce additional oversubscribed cuts beyond Fig. 3b, which can be naturally incorporated as additional big links. Extending Ampel to fundamentally different topologies such as Dragonfly is more difficult, as cuts are no longer disjoint and may grow rapidly. Designing scalable methods to efficiently handle the overlapping cuts remains an open challenge.

References

[1] [n. d.]. Alps | CSCS. https://www.cscs.ch/computers/alps.

[2] [n. d.]. NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/nccl.

[3] [n. d.]. Ultra Ethernet Consortium. https://ultraethernet.org/.

[4] 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 (SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 16–29. doi:10.1145/3230543.3230569

[5] Joao Araujo, Alex Chow, Mark Handley, Ryder Lewis, Christoph Paasch, Jitendra Padhye, Michael Papamichael, Greg Steinbrecher, Amin Tootoonchian, Lihua Yuan, S Anantharamu, Abhishek Dosi, Mohit Garg, Mahdieh Ghazi, Torsten Hoefler, Deepal Jayasinghe, Jithin Jose, Abdul Kabbani, Guohan Lu, Yang Wang, K Doddapaneni, Murali Garimella, Vipin Jain, Yanfang Le, H Nagulapalli, S Narayanan, Rong Pan, Rathina Sabesan, Raghava Sivaramu, Rip Sohan, Eric Davis, Dragos Dumitrescu, Mohan Kalkunte, Bhaswar Mitra, Guglielmo Morandin, Adrian Popa, Costin Raiciu, Eric Spada, John Spillane, Niranjan Vaidya, Aviv Barnea, Idan Burstein, Elazar Cohen, Yamin Friedman, Noam Katz, Masoud Moshref, Yuval Shpigelman, Shahaf Shuler, Shy Shyman, and Sayantan Sur. [n. d.]. Resilient AI Supercomputer Networking Using MRC and SRv6. ([n. d.]).

[6] Tommaso Bonato, Sepehr Abdous, Abdul Kabbani, Ahmad Ghalayini, Nadeen Gebara, Terry Lam, Anup Agarwal, Tiancheng Chen, Zhuolong Yu, Konstantin Taranov, Mahmoud Elhaddad, Daniele De Sensi, Soudeh Ghorbani, and Torsten Hoefler. 2025. Uno: A One-Stop Solution for Inter- and Intra-Data Center Congestion Control and Reliable Connectivity. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25)*. Association for Computing Machinery, New York, NY, USA, 1195–1210. doi:10.1145/3712285.3759884

[7] Tommaso Bonato, Abdul Kabbani, Ahmad Ghalayini, Michael Papamichael, Mohammad Dohadwala, Lukas Gianinazzi, Mikhail Khalilov, Elias Achermann, Daniele De Sensi, and Torsten Hoefler. 2026. REPS: Recycled Entropy Packet Spraying for Adaptive Load Balancing and Failure Mitigation. In *Proceedings of the 21st European Conference on Computer Systems (EUROSYS '26)*. Association for Computing Machinery, New York, NY, USA, 225–246. doi:10.1145/3767295.3769320

[8] Weilin Cai, Juyong Jiang, Le Qin, Junwei Cui, Sunghun Kim, and Jiayi Huang. 2025. Shortcut-Connected Expert Parallelism for Accelerating Mixture of Experts. In *Proceedings of the 42nd International Conference on Machine Learning (ICML'25, Vol. 267)*. JMLR.org, Vancouver, Canada, 6211–6228.

[9] Jiamin Cao, Yu Guan, Kun Qian, Jiaqi Gao, Wencong Xiao, Jianbo Dong, Binzhang Fu, Dennis Cai, and Ennan Zhai. 2024. Crux: GPU-Efficient Communication Scheduling for Deep Learning Training. In *Proceedings of the ACM SIGCOMM 2024 Conference (ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 1–15. doi:10.1145/3651890.3672239

[10] Jiamin Cao, Shangfeng Shi, Jiaqi Gao, Weisen Liu, Yifan Yang, Yichi Xu, Zhilong Zheng, Yu Guan, Kun Qian, Ying Liu, Mingwei Xu, Tianshu Wang, Ning Wang, Jianbo Dong, Binzhang Fu, Dennis Cai, and Ennan Zhai. 2025. SyCCL: Exploiting Symmetry for Efficient Collective Communication Scheduling. In *Proceedings of the ACM SIGCOMM 2025 Conference (SIGCOMM '25)*. Association for Computing Machinery, New York, NY, USA, 645–662. doi:10.1145/3718958.3750499

[11] 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*. ACM, Redmond Washington, 31–36. doi:10.1145/2390231.2390237

[12] Mosharaf Chowdhury and Ion Stoica. 2015. Efficient Coflow Scheduling Without Prior Knowledge. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 393–406. doi:10.1145/2785956.2787480

[13] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. 2014. Efficient Coflow Scheduling with Varys. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*. Association for Computing Machinery, New York, NY, USA, 443–454. doi:10.1145/2619239.2626315

[14] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, Shuqiang Zhang, Mikel Jimenez Fernandez, Shashidhar Gandham, and Hongyi Zeng. 2024. RDMA over Ethernet for Distributed Training at Meta Scale. In *Proceedings of the ACM SIGCOMM 2024 Conference (ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 57–70. doi:10.1145/3651890.3672233

[15] Soudeh Ghorbani, Yimeng Zhao, Srikanth Sundaresan, Ying Zhang, Yijing Zeng, Abhigyan Sharma, Prashanth Kannan, and Cristian Lumezanu. 2025. Congestion Patterns in a Large-scale RDMA Datacenter. In *Proceedings of the 2025 ACM Internet Measurement Conference (IMC '25)*. Association for Computing Machinery, New York, NY, USA, 944–951. doi:10.1145/3730567.3764494

[16] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over Commodity Ethernet at Scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, Florianopolis Brazil, 202–215. doi:10.1145/2934872.2934908

[17] Chenyang Hei, Jiayi Li, Jiamin Cao, Chengxi Gao, Xiuzhu Sha, Tongrui Liu, Dengke Zhang, Ennan Zhai, and Xingwei Wang. 2026. HeteCCL: Synthesizing Near-Optimal Collective Communication Schedules for Heterogeneous GPU Clusters. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. 2533–2551.

[18] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Number 10. Curran Associates Inc., Red Hook, NY, USA, 103–112.

[19] Xin Zhe Khooi, Zhuo Jiang, Pan Xie, Zhigang Cui, Meng Wang, Yuze Jin, Pengfei Huo, Dongyang Wang, Lulu Chen, Lei Wang, Liaoyuan Feng, Xiaodong Liu, Peng Li, Qinlong Wang, Yang Bai, Yongcan Wang, Hao Jin, Jinshuai Sun, Shan Lu, Xiang Shi, Yingkai Zhao, Haiquan Chen, Yi Li, Jianxi Ye, and Mun Choon Chan. 2026. Handling Network Faults in Distributed AI Training: Failover is Now an Option. In *Proceedings of the 21st European Conference on Computer Systems* (McEwan Hall/The University of Edinburgh, Edinburgh, Scotland UK) *(EUROSYS '26)*. Association for Computing Machinery, New York, NY, USA, 263–278. doi:10.1145/3767295.3769322

[20] Yanfang Le, Rong Pan, Peter Newman, Jeremias Blendin, Abdul Kabbani, Vipin Jain, Raghava Sivaramu, and Francis Matus. 2024. STrack: A Reliable Multipath Transport for AI/ML Clusters. https://arxiv.org/abs/2407.15266v2.

[21] Wenxue Li, Xiangzhou Liu, Yuxuan Li, Yilun Jin, Han Tian, Zhizhen Zhong, Guyue Liu, Ying Zhang, and Kai Chen. 2024. Understanding Communication Characteristics of Distributed Training. In *Proceedings of the 8th Asia-Pacific Workshop on Networking (APNet '24)*. Association for Computing Machinery, New York, NY, USA, 1–8. doi:10.1145/3663408.3663409

[22] Xudong Liao, Yijun Sun, Han Tian, Xinchen Wan, Yilun Jin, Zilong Wang, Zhenghang Ren, Xinyang Huang, Wenxue Li, Kin Fai Tse, Zhizhen Zhong, Guyue Liu, Ying Zhang, Xiaofeng Ye, Yiming Zhang, and Kai Chen. 2025. MixNet: A Runtime Reconfigurable Optical-Electrical Fabric for Distributed Mixture-of-Experts Training. In *Proceedings of the ACM SIGCOMM 2025 Conference (SIGCOMM '25)*. Association for Computing Machinery, New York, NY, USA, 554–574. doi:10.1145/3718958.3750465

[23] Tongrui Liu, Chenyang Hei, Fuliang Li, Chengxi Gao, Jiamin Cao, Tianshu Wang, Ennan Zhai, and Xingwei Wang. 2025. ResCCL: Resource-Efficient Scheduling for Collective Communication. In *Proceedings of the ACM SIGCOMM 2025 Conference (SIGCOMM '25)*. Association for Computing Machinery, New York, NY, USA, 55–70. doi:10.1145/3718958.3750514

[24] Xuting Liu, Behnaz Arzani, Siva Kesava Reddy Kakarla, Liangyu Zhao, Vincent Liu, Miguel Castro, Srikanth Kandula, and Luke Marshall. 2024. Rethinking Machine Learning Collective Communication as a Multi-Commodity Flow Problem. In *Proceedings of the ACM SIGCOMM 2024 Conference (ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 16–37. doi:10.1145/3651890.3672249

[25] Jie Lu, Jiaqi Gao, Fei Feng, Zhiqiang He, Menglei Zheng, Kun Liu, Jun He, Binbin Liao, Suwei Xu, Ke Sun, Yongjia Mo, Qinghua Peng, Jilie Luo, Qingxu Li, Gang Lu, Zishu Wang, Jianbo Dong, Kunling He, Sheng Cheng, Jiamin Cao, Hairong Jiao, Pengcheng Zhang, Shu Ma, Lingjun Zhu, Chao Shi, Yangming Zhang, Yiquan Chen, Wei Wang, Shuhong Zhu, Xingru Li, Qiang Wang, Jiang Liu, Chao Wang, Wei Lin, Ennan Zhai, Jiesheng Wu, Qiang Liu, Binzhang Fu, and Dennis Cai. 2025. Alibaba Stellar: A New Generation RDMA Network for Cloud AI. In *Proceedings of the ACM SIGCOMM 2025 Conference (SIGCOMM '25)*. Association for Computing Machinery, New York, NY, USA, 453–466. doi:10.1145/3718958.3750539

[26] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. 2020. Themis: Fair and Efficient GPU Cluster Scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 289–304.

[27] Qingkai Meng, Hao Zheng, Zhenhui Zhang, ChonLam Lao, Chengyuan Huang, Baojia Li, Ziyuan Zhu, Hao Lu, Weizhen Dang, Zitong Lin, Weifeng Zhang, Lingfeng Liu, Yuanyuan Gong, Chunzhi He, Xiaoyuan Hu, Yinben Xia, Xiang Li, Zekun He, Yachen Wang, Xianneng Zou, Kun Yang, Gianni Antichi, Guihai Chen, and Chen Tian. 2025. Astral: A Datacenter Infrastructure for Large Language Model Training at Scale. In *Proceedings of the ACM SIGCOMM 2025 Conference (SIGCOMM '25)*. Association for Computing Machinery, New York, NY, USA, 609–625. doi:10.1145/3718958.3750521

[28] 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 (SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 1–15. doi:10.1145/3341301.3359646

[29] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, Chao Wang, Peng Wang, Pengcheng Zhang, Xianlong Zeng, Eddie Ruan, Zhiping Yao, Ennan Zhai, and Dennis Cai. 2024. Alibaba HPN: A Data Center Network for Large Language Model Training.

In *Proceedings of the ACM SIGCOMM 2024 Conference (ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 691–706. doi:10.1145/3651890.3672265

[30] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. 2021. Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*.

[31] Costin Raiciu, Christopher Pluntke, Sebastien Barre, Adam Greenhalgh, Damon Wischik, and Mark Handley. 2010. Data Center Networking with Multipath TCP. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (Hotnets-IX)*. Association for Computing Machinery, New York, NY, USA, 1–6. doi:10.1145/1868447.1868457

[32] Sudarsanan Rajasekaran, Manya Ghobadi, and Aditya Akella. 2024. CASSINI: Network-Aware Job Scheduling in Machine Learning Clusters. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 1403–1420.

[33] Sudarsanan Rajasekaran, Sanjoli Narang, Anton A. Zabreyko, and Manya Ghobadi. 2024. MLTCP: A Distributed Technique to Approximate Centralized Flow Scheduling For Machine Learning. In *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks (HotNets '24)*. Association for Computing Machinery, New York, NY, USA, 167–176. doi:10.1145/3696348.3696878

[34] 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*. 1–16. doi:10.1109/SC41405.2020.00024

[35] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. 2023. TACCL: Guiding Collective Algorithm Synthesis Using Communication Sketches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 593–612.

[36] Siyuan Shen, Tommaso Bonato, Zhiyi Hu, Pasquale Jordan, Tiancheng Chen, and Torsten Hoefler. 2025. ATLAHS: An Application-centric Network Simulator Toolchain for AI, HPC, and Distributed Storage. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25)*. Association for Computing Machinery, New York, NY, USA, 349–367. doi:10.1145/3712285.3759838

[37] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv:1909.08053 [cs] doi:10.48550/arXiv.1909.08053

[38] Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S. Rellermeyer. 2020. A Survey on Distributed Machine Learning. *ACM Comput. Surv.* 53, 2 (March 2020), 30:1–30:33. doi:10.1145/3377454

[39] Xinchen Wan, Xinyu Yang, Kaiqiang Xu, Xudong Liao, Yilun Jin, Yijun Sun, Zhenghang Ren, Han Tian, and Kai Chen. 2025. Coflow Scheduling for LLM Training. In *Proceedings of the ACM SIGCOMM 2025 Conference (SIGCOMM '25)*. Association for Computing Machinery, New York, NY, USA, 1232–1234. doi:10.1145/3718958.3750467

[40] Weiyang Wang, Manya Ghobadi, Kayvon Shakeri, Ying Zhang, and Naader Hasani. 2024. Rail-Only: A Low-Cost High-Performance Network for Training LLMs with Trillion Parameters. In *2024 IEEE Symposium on High-Performance Interconnects (HOTI)*. 1–10. doi:10.1109/HOTI63208.2024.00013

[41] Weiyang Wang, Moein Khazraee, Zhizhen Zhong, Manya Ghobadi, Zhihao Jia, Dheevatsa Mudigere, Ying Zhang, and Anthony Kewitsch. 2023. TopoOpt: Co-optimizing Network Topology and Parallelization Strategy for Distributed Training Jobs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 739–767.

[42] Yihao Zhao, Yuanqiang Liu, Yanghua Peng, Yibo Zhu, Xuanzhe Liu, and Xin Jin. 2022. Multi-Resource Interleaving for Deep Learning Training. In *Proceedings of the ACM SIGCOMM 2022 Conference (SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 428–440. doi:10.1145/3544216.3544224

A Appendix

In this section, we present the Network-Cut Coflow Scheduling (NCCS) problem formulation.

A.1 Problem Formulation

We formulate the offline version of the Network-Cut Coflow Scheduling (NCCS) problem as an objective function and a set of constraints. The formulation we provide is built upon the Coflow Scheduling formulation described in Varys [13], though it presents some important differences.

Every coflow is a set of flows from various sources to various destinations. The constant d_{ij}^k defines the amount of data that coflow k wants to send from source i to destination j. The variable $r_{ij}^k(t)$ is the amount of data sent from source i to destination j for coflow k at time t. The variable C_k is the completion time of coflow k.

In addition to these values, we also define a set of oversubscribed network cuts as defined in 3.1. In particular, each cut is represented as two constraints, i.e. in both directions (for example, incoming and outgoing bandwidth for a pod). The capacity of a cut c is defined by the constant a_c. By the notation $\forall(i,j) \vDash c$ we refer to all the source-destination pairs whose paths cross c.

Note that, for this formulation to be correct, the oversubscribed network cuts must *also* include the incoming and outgoing links of each NIC. Intuitively, such a link is an oversubscribed cut as the host could potentially generate unlimited demand, but is bound by the link's bandwidth.

$$\min \quad \sum_{k=1}^{K} C_k \tag{1}$$

$$\text{s.t.} \quad \sum_{k,(i,j)\vDash c} r_{ij}^k(t) \le a_c \qquad \forall t, \forall c \tag{2}$$

$$\sum_{t=1}^{C_k} r_{ij}^k(t) \ge d_{ij}^k \qquad \forall i, j, k \tag{3}$$

Inequality (2) represents the capacity constraints: the amount of data sent at a given time over a certain bottleneck should not exceed the bottleneck's capacity. Inequality (3) ensures that the duration of a coflow is as long as its slowest flow. Note that this formulation does not constitute an ILP, and cannot be directly used in an optimizer.

The crucial difference between NCCS and classical coflow scheduling formulation shown in Varys is that a flow may now be subject to several bottlenecks rather than just sender and receiver NICs.

The problem is NP-hard. One can reduce the classical Coflow Scheduling problem into the Bottleneck Coflow Scheduling problem by selecting the oversubscribed cuts to be only the incoming and outgoing links of each host or NIC. Because the classical Coflow Scheduling problem is NP-hard, so is our variation.