Towards Domain-Specific Network Transport for Distributed DNN Training

Hao Wang and Han Tian, *iSING Lab, Hong Kong University of Science and Technology;* Jingrong Chen, *Duke University;* Xinchen Wan, Jiacheng Xia, and Gaoxiong Zeng, *iSING Lab, Hong Kong University of Science and Technology;* Wei Bai, *Microsoft;* Junchen Jiang, *University of Chicago;* Yong Wang and Kai Chen, *iSING Lab, Hong Kong University of Science and Technology*

https://www.usenix.org/conference/nsdi24/presentation/wang-hao

This paper is included in the Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation.

April 16–18, 2024 • Santa Clara, CA, USA

978-1-939133-39-7

Towards Domain-Specific Network Transport for Distributed DNN Training

Hao Wang[1], *Han Tian*[1], *Jingrong Chen*[2], *Xinchen Wan*[1], *Jiacheng Xia*[1], *Gaoxiong Zeng*[1]
Wei Bai[3]*, *Junchen Jiang*[4], *Yong Wang*[1], *Kai Chen*[1]
[1]*iSING Lab, Hong Kong University of Science and Technology*
[2]*Duke University*, [3]*Microsoft*, [4]*University of Chicago*

Abstract

Machine learning (ML) applications present rich characteristics to underlying network transport, yet little work has been done so far to systematically exploit these properties in transport design. This paper takes the initiative to pursue a domain-specific network transport, called MLT[1], for distributed DNN training that fully embraces several unique characteristics of machine learning.

At its heart, MLT employs three simple-yet-effective techniques to form a 3-step progressive scheme against long tail latency caused by transient packet drops and queueing. First, it leverages the *independencies among gradient updates* to enable per-packet load balancing to minimize network hotspots without worrying about packet re-ordering. Then, if hotspot arises, it performs priority queueing/dropping based on the *layers and magnitudes of gradients* to optimize the model convergence. Lastly, if drop occurs, it enables *bounded-loss tolerance*—a certain amount of gradient losses tolerated by the DNN training without affecting the model accuracy. MLT is readily deployable with commodity switches and imposes minimal modifications on various DNN training libraries (e.g., TensorFlow, MXNet and PyTorch) and communication routines (e.g., PS and Ring All-reduce). We show, via both testbed experiments and simulations, that MLT effectively optimizes network tail latency and delivers up to 62.2% better end-to-end training performance over prior work.

1 Introduction

Deep Neural Networks (DNNs) have been paramount to modern machine learning applications in computer vision (CV) [41, 57, 91] and natural language processing (NLP) [74, 99, 102]. However, training DNNs can be notoriously slow, due importantly to the sheer volumes of gradients that need to be frequently shuffled. While individual forward/backward propagations can be massively parallelized, typical DNN training still includes 100s of iterations, each ending up with shuffling massive gradients across 10s to 100s of workers, potentially causing severe worst-case congestion and tail latencies and slowing DNN training down to a crawl. Such communication bottlenecks have been witnessed in production clusters and reported in many literatures [33, 39, 50, 69, 78, 81, 100, 114].

To reduce the communication overhead in distributed DNN training, many solutions have been proposed recently. Some of them attempt to reduce traffic volume at application layer through gradient compression (e.g., sparsification [33, 66, 101] or quantization [12]), while others seek to overlap communication and computation through tensor partitioning and transmission scheduling [39, 49, 81]. However, while these solutions effectively optimize the *average* flow completion time (FCT) in general, they are susceptible to transient queueing or loss which could lead to severe *tail* FCT when network is under pressure. As we show in §2.2, by reducing the total traffic volume, gradient compression can reduce the average FCT, but fails to avoid the long tail FCT caused by sporadic packet loss in the network, which adversely affects the DNN training performance.

One plausible way to tackle this problem is to borrow the sophisticated datacenter network transport solutions such as DeTail [110], pFabric [15], Homa [77] and NDP [38], to name a few, to deep learning clusters. However, while these advanced solutions can potentially deliver near-optimal average or tail latencies, they are too complex by either re-factoring the whole network stack from physical layer, or requiring the support of specific switch ASICs (e.g., cutting payload [38]), or assuming non-blocking network cores, making them hard to deploy in practice.

In this paper, we ask: *can we design a simple, effective yet readily-deployable solution to the above problem?* Motivated by the special properties of machine learning (§3), we answer the question affirmatively by presenting MLT (§4), a domain-specific network transport that can optimize both average and tail FCTs of distributed DNN training by exploiting a series of ML-specific properties, in a progressive manner.

First, MLT exploits the property of inter-packet independency to perform per-packet load balancing, which minimizes network hotspots. Per-packet load balancing is highly desirable by almost all prior datacenter solutions with multi-pathing. However, none could really enjoy its benefit without paying considerable cost for packet re-ordering, with some of them backing off to flowlet as a compromise [13, 96]. Unlike traditional applications where one message often contains multiple packets (thus order-dependent), we note that messages in DNN training are essentially gradients, which are small enough (32bits or less) to allow multiple messages packed in one packet (thus order-free) (§3.1). By exploring this, MLT can effectively minimize network hotspots with perfect packet spraying without re-ordering concern.

Second, when hotspot arises, MLT performs priority

*Now with NVIDIA
[1]**MLT**: **M**achine **L**earning **T**ransport for AI-centric Networking (AICN).

queueing or dropping at the switch based on the importance of gradients in the packets. As elaborated in §3.3, we find that the importance of a gradient depends on its position (i.e., DNN layer) and magnitude. Typically, gradients of front layers are less important than back layers due to the popular pre-training technique used in CV and NLP applications: the front layers are often pre-trained on large/generic datasets and thus need fewer updates than the back layers [107]. Meanwhile, larger gradients are more important than smaller ones, because large gradients are critical for identifying the correlation between the intermediate features and the final prediction result.

Lastly, if packet drop occurs, MLT resorts to its final defense—bounded-loss tolerance—no retransmission up to a certain packet loss fraction allowed by the *approximate* SGD-based training, to protect against the long tail latency (§3.2). In contrast to today's all-or-nothing transport primitive (TCP or UDP), MLT supports a bounded-loss tolerance transport primitive: by intentionally ignoring packets (bounded by p) delayed or lost in the network without retransmission, MLT effectively cuts tail latency while still maintaining the same model accuracy. We note that while such loss-tolerance property has been used in prior work to reduce traffic volume [101] at the application layer which reduces the average FCT, we reuse it at the transport layer to cut the tail (§4).

We have implemented MLT (§5) with commodity switches, integrated it with TensorFlow, MXNet and PyTorch, and deployed it over our small-scale testbed with 64 RTX3090 GPUs. In our implementation, we only use basic switch functions such as ECN/RED and priority queueing, and build MLT transport protocol in user space without modifying kernel network stack. Through extensive testbed experiments and large-scale simulations (§6), we find that:

- Compared to the state-of-the-art in communication optimization of DNN training (BytePS [52]), MLT achieves up to 12.0%–62.2% training speedup across various DNN models while maintaining the same accuracy (§6.1).
- Compared to the PyTorch FSDP [115] in fine-tuning Large Language Models (LLMs) with fully sharded data parallelism [115], MLT achieves up to 35.5% speedup without affecting the fine-tuning process (§6.1).
- Compared to one of the best datacenter transports (NDP [38]), MLT delivers comparable network performance (in fact, 6.7%/10.3% lower average/tail FCTs) under realistic ResNet50-induced traffic, with no switch modification (§6.2).
- Each design component of MLT contributes effectively to the ultimate performance of MLT (§6.3).

We believe MLT showcases a first step towards an AI-centric networking design that systematically explores a series of ML-specific characteristics. While many questions, from theoretical understanding of more ML algorithmics/characteristics to practical real-world implementation/deployment, remain open (§7), our exploration in MLT has shown early promise

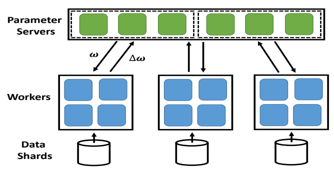


Figure 1: Data parallelism with Parameter Server

of this direction and hopefully could inspire this community to think more on how to design AI-centric networking for ML applications by exploiting its special properties.

2 Distributed DNN Training

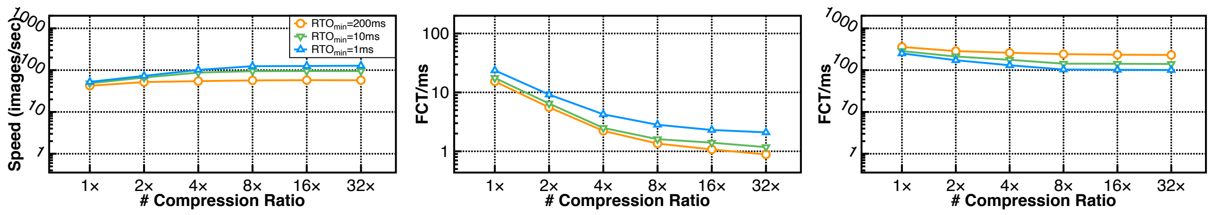
2.1 Communication Overhead

Training basics: DNNs learn intricate representations on large datasets. A training process refines a DNN model upon a dataset for many epochs, each consisting of multiple iterations. In each iteration: (1) The DNN model and a data partition (or mini-batch) are taken as input; (2) the mini-batch travels through the model from the first layer to the last layer, called *forward propagation (FP)*, and computes a loss; (3) with the loss derived, it computes the gradients backwards from the last layer to the first layer, called *backward propagation (BP)*; and finally (4) the gradients, which represent information acquired from the mini-batch, are used to update the model with optimization algorithms, normally Stochastic Gradient Descent (SGD) [21, 54, 82].

Parallelism schemes: There are mainly two types of parallelism schemes for distributed ML: Data Parallelism (DP) and Model Parallelism (MP). DP aims to faster batch processing speed while MP tries to achieve better memory efficiency. When a single GPU can accommodate a model, the mainstream approach is DP [52]. With the rise of LLMs, even the latest H100 [9] GPU cannot fit an entire model. As a result, researchers start to use MP. Some leverage pipelining to enhance computational efficiency (pipeline parallelism [78]), while others further divide a tensor across multiple GPUs (tensor parallelism [90]). Nevertheless, recent wisdom, e.g. , ZeRO [84] and FSDP [115], eliminates memory redundancies of DP and achieves comparable or better performance than MP in terms of memory efficiency, especially for models that are hard to evenly distribute [84]. Such DP approach can even achieve super-linear speedup [84].

Therefore, in this paper, we focus on data parallelism and leave others as future work (§7). To speed up with DP, mini-batches of training data are distributed across multiple machines (or workers) as shown in Figure 1. Different workers share the same global model and independently compute gradients (with FP and BP) on their respective mini-batches. Then, gradients from all workers are synchronized and aggregated to update the global model[2], using either Parameter Server

[2]There are three general approaches to model synchronization: Bulk Synchronous Parallelism (BSP) [95], Asynchronous Parallel (ASP) [86], and Stale Synchronous Parallel (SSP) [104]. Among them, BSP, in which all workers need to train on the same iterations, is preferred in production due



(a) Training Speed (for one worker) (b) Average Flow Completion Time (c) Tail Flow Completion Time

Figure 2: Time cost of training GoogleNet on ImageNet with different compression ratios and RTO_{min}s in our testbed. Note that the minimum RTO_{min} in Linux kernel is 1ms, and we also deep-dive microsecond level RTO_{min} with simulations in §6.3. However, please note that the realistic RTO_{min} in production is set typically at millisecond level even though the base RTT could be at microsecond level, mainly due to the reason explained in Footnote-3.

architecture [60] or collective routines like All-reduce [80]:

- *Parameter Server (PS)* [60] is a logically centralized key-value store. In each iteration, workers pull new model parameters from PS for training, and then push gradients to PS for model updating. PS enables flexible parameter synchronization pattern and is generally fault-tolerant.
- *All-reduce* [80] is a collective operation to sum up gradients of all workers. A popular implementation is Ring All-reduce [88], in which workers form a logical ring. Each worker receives a chunk of gradients from one neighbor, adds to its local copy, and sends the chunk to the other neighbor, until all the gradients are aggregated. Compared to PS, Ring All-reduce generates more uniform traffic pattern, but has higher synchronization overhead due to the smaller flow size and longer communication chain [27].

Communication overhead: The above gradient transmission and model synchronization demand efficient network transport [81]. In each iteration, each worker can send and receive model gradients with tens to thousands of MBs [41,57,91]. As a result, the network communication often takes a significant portion of the total training time. This overhead has been observed by many recent literatures [31, 33, 39, 50, 60, 63, 78, 81, 112]. For example, training AlexNet on 8 nodes requires more than 26Gbps bandwidth to avoid blocking computation of next iteration [112]. Recent measurement has also shown that communication accounts for 90% of the total training time over 32 GPUs [78]. Furthermore, as reported by a large online service provider, due to the communication overhead, the training performance is far from linear speed-up with an increasing number of GPU servers in many of their internal and publicly available training workloads [81].

2.2 Existing Solutions and Problems

There exist solutions to the above problems from different angles, however, they all have shortcomings.

Gradient sparsification and quantization: One direct way to optimize the communication overhead is to reduce traffic volume at application layer through gradient compression (e.g., sparsification [33, 66, 101] or quantization [12]). Specifically, gradient sparsification reduces network traffic by filtering near zero gradients, whereas gradient quantization represents the gradients with lower-precision floating point numbers to reduce traffic volume. While both approaches reduce overall traffic volume, they do not make communication completely immune to long-tail latency due to packet drops or queueing from micro-burstiness.

To show the problem, we train GoogleNet [93], a widely used model, on ImageNet [29] dataset with different compression ratios from 1× (original) to 32× (93.8%). Our testbed contains 64 RTX3090 GPUs connected by 25Gbps bandwidth. The detailed setup is in §6.1.1. For training, we adopted the PS architecture and set the number of servers equal to that of workers [81]. During the communication, workers send gradient updates simultaneously to workers or PSes, which may elicit a pathological phenomenon called incast [24] congestion. As RTO_{min}, short for minimum retransmission timeout, matters for the impact of incast [24], we conduct the experiments with different RTO_{min} values: 200*ms*—the default value for Linux, 10*ms*—a common setting of previous works [25], 1*ms*—the minimum setting for Linux (release version) [24].[3] To exclude the overhead of gradient compression operations, we simulate the compression process by directly modifying the gradient's length. As reference, the computation time of GoogleNet is about 30*ms* for a worker in one iteration. To emulate a multi-job environment, we also run two concurrent background DNN training tasks.

Figure 2 shows the experimental results. One key observation is that the improvement on training speed is quite limited even with a significant gradient compression (Figure 2(a)), e.g., a 32× compression in traffic volume only translates to 1.36-2.4× improvement in training speed. The root cause behind is shown in Figure 2(b)(c): while the gradient compres-

to its simplicity and convergence properties over ASP/SSP [11, 35, 72, 78]. Furthermore, BSP also produces deterministic and reproducible results, which are needed in hyper-parameter tuning [103]. For these reasons, we focus on BSP in this paper and leave ASP/SSP for future exploration.

[3]For theoretical analysis, we also simulate the microsecond level RTO_{min} in §6.3, however, in practice, RTO_{min} in production clusters is typically set at millisecond level to avoid spurious retransmissions under large queue-buildups. Today's commodity switches use shared buffer management to improve burst tolerance, thus causing high queueing delay. For example, Broadcom Trident II [116] with 12MB shared buffer and 32 40G ports can cause up to $12MB/40Gbps = 2.4ms$ queueing delay per hop.

sion reduces the average FCT steadily from 15.3ms to 1.1ms, the tail FCT, which decides the ultimate communication time of one iteration, remains high. For large $RTO_{min} = 200ms$, one retransmission timeout costs at least 200ms delay, so the tail FCT is above 200ms. For smaller $RTO_{min} = 10ms$ or 1ms, we find that the actual RTO is not determined by RTO_{min}, and our measurement shows that the actual RTO is 48ms in case of 32$\times$ compression, which is largely caused by consecutive losses and spurious retransmissions. Consider the $\sim$30ms computation time, such long-tail latency compromises the overall training efficiency, undermining the benefit brought by gradient compression.

The reader may wonder if operators can deploy priority flow control (PFC) [1] to eliminate congestion loss, thus mitigating long tail latency shown above. However, many studies have shown that PFC causes a series of performance (e.g., congestion spreading and unfairness [116]) and management problems (e.g., PFC storm and even deadlock [36]). In a PFC-enabled network, a flow can be paused due to congestion that is not even on its path. Moreover, PFC cannot eliminate packet losses due to link failure, including both fail-stop failure and gray failure [48]. Essentially, the head-of-line blocking nature of PFC makes the network difficult to understand and manage, and we leave it for future exploration.

Computation/communication overlapping and scheduling: Most DNN training frameworks (e.g., MXNet, PyTorch and TensorFlow) and Poseidon [112] overlap communication with backward propagation (BP). Instead of waiting for BP on all DNN layers, they send gradients as soon as one DNN layer is processed, pipelining the gradient transmission of some layers with the gradient computation of other layers. Building on Poseidon, P3, TicTac [39] and ByteScheduler/BytePS [52, 81] further overlap communication of the current iteration with the forward propagation (FP) of the next iteration through tensor partitioning and priority-based transmission scheduling. Despite being helpful, these solutions are still insufficient because, by design, such DNN structure-level overlappings do not directly solve the tail latency issue shown in Figure 2. Furthermore, they ignore the network as they are purely endhost-based solutions, but network switches are unaware of such application-level priorities when choosing which packets to queue or drop.

Advanced datacenter transport: One plausible way to tackle the tough tail latency could be introducing the sophisticated datacenter network transport solutions such as DeTail [110], pFabric [15], Homa [77] and NDP [38], etc., to deep learning clusters. These deliberately designed solutions can potentially deliver near-optimal average or tail latencies, which may effectively solve the issues in Figure 2. However, the downside is that these solutions are typically too complex, e.g., either re-factoring the whole network stack from physical layer, or requiring switch hardware modifications (e.g., cutting payload [38]), or assuming non-blocking network cores, making them hard to deploy in practice.

3 Observations and Opportunities

We seek an effective yet readily-deployable solution without much complexity. By exploiting the domain-specific properties of DNN training, we make the following observations which provide insights for our design.

3.1 Packets Are Order-Independent

Unlike many other applications, packets in DNN training can tolerate out-of-order delivery. This is because in traditional applications, the application data unit or message usually spans multiple packets, and therefore, ordering needs to be maintained among packets of a message. In contrast, in DNN training context, the message is just a gradient or parameter, which is typically represented as a 32-bit floating pointer number [60]. Therefore, multiple message can be packed within one packet, and packets can be interpreted independently. Such inter-packet order-independency provides an opportunity for packet-level load balancing in the network.

Meanwhile, the traffic in DNN training is predictable. To enhance communication efficiency, DNN training frameworks divide a tensor into buckets, e.g., the default maximum bucket size in PyTorch [79] is 25MB. To better overlap communication with computation, BytePS [52] implements tensor partitioning with a default size of 4MB. Such pre-determined maximum bucket or partition sizes facilitate tensor re-construction at the receiver end.

3.2 Packet Losses Are Bound-Tolerant

SGD-based DNN training is essentially an *approximation algorithm* which estimates better parameter values based on gradients calculated from mini-batches [21, 62]. It can tolerate loss for two main reasons: (1) gradient loss from workers results in dynamic mini-batch sizes instead of directly causing errors, as SGD updates the model with the average gradient values collected from workers, the expected gradient for update can be unbiased under certain random loss; (2) even if an error occurs, it will self-heal automatically. This is because in each round SGD recalibrates the gradient vector towards the optimal based on the current model weights, errors caused by loss in the earlier iterations will not be propagated to the latter ones. Recent work [108] has theoretically proven that, under random loss, this algorithm can converge within the same magnitude of iterations.

Before using this property, we try to understand it more precisely. We simulate the scenario of distributed SGD on different DNN models, and randomly set the gradients to be 0 with ratio p to simulate packet losses in the network. To quantify the impact, we first train the baseline (no loss) for 500 iterations, which is large enough to make the test accuracy almost constant for all the models and datasets. The final test accuracy value is set as Quality Target [4]. Then, we measure the rounds needed to reach the Quality Target for each ratio p. This approach is the same as MLPerf [4], a well-known ML benchmark. We repeat each measurement for 10 times

Bounded-loss ratio (p)	0%-1%		1%-2%	>2%
Model	GoogleNet [93] (0.7%)	LSTM [42] (0.8%)	ResNet34 [41] (1%)	Wide ResNet50 [109] (2.4%)
	AlexNet [57](0.8%)	VGG16 [91] (0.8%)	GRU [26] (1.2%)	ResNet50 [41] (2.4%)
	EfficientNetB0 [94] (0.8%)	VGG19 [91] (0.9%)	Wide ResNet101 [109] (1.3%)	ShuffleNetV2 [71] (2.5%)
	VGG13 [91] (0.9%)	ResNet18 [41] (0.9%)	DenseNet169 [47] (1.6%)	ResNet101 [41] (3.3%)

Table 1: The bounded-loss tolerance property for a wide range of DNN models. For LSTM and GRU models, we train them as NLP tasks using Wikitext-2 [75]; Other models are trained as CV tasks using ImageNet100 [7].

Model	GoogleNet		GRU	
Dataset	Cifar100	Caltech101	Wiki.-2	Wiki.-103
Grad. loss	0.8%	0.8%	1.2%	1.4%

Table 2: Loss-tolerant bounds for different datasets

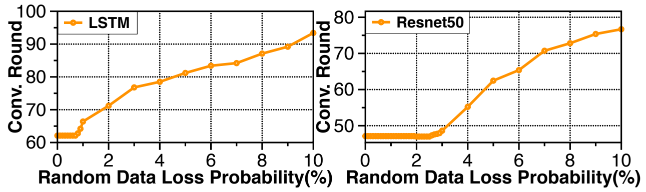


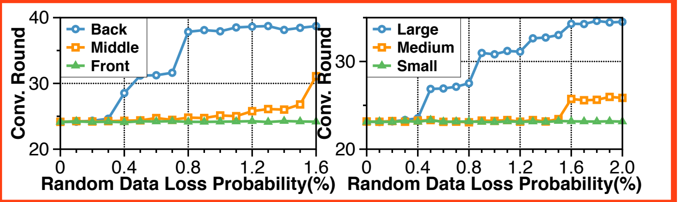
Figure 3: Impact of loss on model convergence: when the loss ratio is below 0.8% (for LSTM) and 2.5% (for ResNet50), the models converge with the same rounds to the same accuracies, indicating loss-tolerance.

and calculate the average. Figure 3 shows the example results of LSTM [42] and ResNet50 [41] trained on Wikitext-2 [75] and Cifar100 [56] datasets. We find that: (1) when the loss ratio is below 10%, we can achieve the Quality Target; and (2) when the loss ratio is below 0.8% (for LSTM) and 2.5% (for ResNet50), we can achieve this with *exactly the same rounds*. We further validate this feature across a wide range of DNN models using several general training datasets. Table 1 summarizes the results for these models, achieving the Quality Target with the same rounds.

We refer to this feature as *bounded-loss tolerance*: DNN training can tolerate a certain fraction p of data loss while still converging with the same iterations to the same accuracy. Despite our fine-grained profiling above, we note that the general property of tolerating gradient loss in DNN training is not new and has been explored in prior works [44, 101, 108]. However, while these existing solutions leverage it for reducing traffic volume [101] or designing reliable PS algorithms [108] at application layer to reduce average FCT, our paper reuses it at network transport layer to cut the tail (§4).

In addition, while different models may have different loss tolerance bounds, we observe that the bounds of one model for different datasets we used remain similar. We show a case in Table 2, in which the bounds for GoogleNet over Cifar100 [56] and Caltech101 [34] are the same, while the bounds for GRU over Wikitext-2 and Wikitext-103 [75] only differ by 0.2%. This enables us to profile the loss tolerance bound values for general DNN models[4].

[4]We note that in practice the bound of a model may vary if datasets differ greatly in some aspects. To explore the loss tolerance bound of a model on a large dataset, one practical way is to use the tolerance bound derived from a smaller sampled sub-dataset from the original dataset as an approximation. Through experiments, we find that the loss tolerance bounds remain almost the same between the sampled sub-dataset and the original dataset. We leave a full exploration of this as future work.



(a) Loss on different layers (b) Loss with different values

Figure 4: Gradient losses on different NN layers (a) and with different magnitudes (b) generate different impacts on model convergence.

3.3 Packets/Gradients Differ in Importance

We observe that different gradients have different impacts on DNN training, depending on their layer positions in DNNs [111], and their magnitudes [44].

First, for convergence, gradients of front layers are often *less important* than back layers due to the prevalent pre-training technique used in CV and NLP applications. Taking CV as an example, front layers extract low-level features such as edges and corners, whereas back layers learn more complex concepts like shapes of certain object. Due to the generality of low-level features across different tasks and datasets, people often pre-train the front layers of their DNNs on large and generic datasets, e.g., ImageNet [29], and then fine-tune them over the target dataset, which will accelerate the training and improve the model performance [107]. With per-training, the front layers extracting low-level features are generally well learned, thus need fewer updates than the back layers. Meanwhile, gradients of front layers are more *urgent* when pipelining strategies [50, 81] are employed, because the FP can start once the front-layer gradients are received.

To illustrate the impact of gradient loss on different layers, we train ResNet50 on Cifar100 [56] during which we randomly discard gradients from: the front layers (the first 20% layers), the middle layers (the middle 20% layers), and the back layers (the last 20% layers) with varying loss probabilities. We first trained the model without discarding gradients and found that the test accuracy converged to 93%, thus in our experiment we set 93% as the target accuracy. Results in Figure 4(a) show that front-layer gradients have a much higher bounded-loss tolerance than back-layer gradients. For instance, to maintain the same convergence speed and accuracy, we can only tolerate 0.3% gradient loss from the back layers but over 5% from the front layers.

Second, larger gradients are typically more *important* than smaller ones. This is because larger gradients are more effective for SGD to identify the correlations between the features and the task than that of smaller ones. As a result, their losses have more negative impact on model accuracy [59]. Furthermore, larger gradients indicate bigger learning step sizes, thus having more impact on convergence speed.

To show the impact of gradient loss of different magnitudes, we again consider three scenarios: randomly dropping gradients among the smallest 20%, the medium 20%, and the largest 20% magnitudes with different loss probabilities. Results in Fig 4(b) show that dropping larger gradients has a much lower bounded-loss tolerance than that of dropping smaller ones. For example, to maintain the same convergence property, we can tolerate more than 20% loss of small gradients but only 0.4% of the large gradients.

4 The MLT Design

MLT is inspired by the above observations. In this section, we first introduce the key ideas of MLT (§4.1), and then present the detailed mechanisms (§4.2). We theoretically prove the convergence of MLT in Appendix A.

4.1 Key Ideas

1. Minimizing hotspots with *order-free* per-packet load balancing. Load balancing aims to eliminate hotspots by spreading traffic onto multiple paths. Ideally, it should be done at packet-level. However, to avoid costly packet reordering, current practice remains to work at the sub-optimal coarse granularity [13, 15, 19, 113]. Based on the observation in §3.1, packets of DNN training are order-free, which enables packet-level spreading without reordering concerns. Thus, MLT employs per-packet load balancing to minimize network hotspots (§4.2.1).

2. Gradient-aware packet queueing and dropping. While DNN training tolerates certain packet losses, the impact of losing different gradients differs as per the observation in §3.3: 1) in terms of DNN layer, front layer gradients are less important than back layer ones, whose dropping has less impact on model convergence; 2) in terms of magnitude, large gradients are more important than small ones, whose dropping has more impact on model accuracy. To respect them, when the switch queue is full and some packets have to be dropped, MLT enforces a gradient-aware selective dropping: 1) Packets carrying front layer gradients will be selected for dropping over those carrying back layer ones; 2) Packets carrying smaller gradients will be selected for dropping over those carrying larger ones.

Furthermore, as mentioned in §3.3, while gradients of front layers are less important for convergence, they are more urgent for training pipelining, since FP can start as soon as the front-layer tensors are received [81]. Therefore, in addition to selective dropping, we further enforce priority queueing to prioritize front-layer packets (§4.2.2).

3. Cutting tail latency with bounded-loss tolerance. As shown in §2.2, gradient compression does not completely solve the long tail latency issue. This is because network congestion (packet losses or queueing) may be caused not

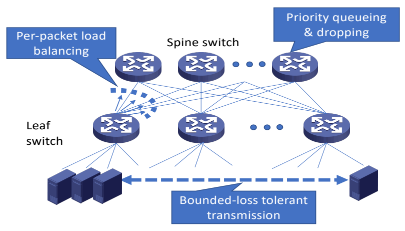

Per-packet load balancing
Spine switch
Priority queueing & dropping
Leaf switch
Bounded-loss tolerant transmission

Figure 5: MLT Overview

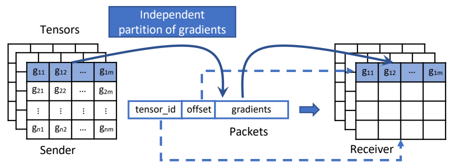

Tensors
Independent partition of gradients
g_{11} g_{12} ··· g_{1m}
g_{21} g_{22} ··· g_{2m}
g_{n1} g_{n2} ··· g_{nm}
tensor_id
offset
gradients
Packets
Sender
Receiver

Figure 6: Tensor packing and unpacking

only by traffic volume, but also by traffic pattern, e.g., high fan-in burst in a short time.

MLT exploits bounded-loss tolerance (§3.2) to address the tail latency. Currently, reliability in transport control is "all-or-nothing": TCP requires all packets to be received and thus may get blocked by a tiny fraction of packet losses waiting for retransmission; whereas UDP has no reliability guarantee. Neither suits for DNN training. Instead, MLT uses a bounded-loss tolerant transport protocol that tolerates up to a bound p fraction of packet losses without retransmission, thus effectively cutting the tail latency while not degrading the training outcome (§4.2.3).

4.2 Mechanisms

Read every word under this section, even not highlighted

Figure 5 overviews MLT, with a 3-step workflow. First, data traffic is spread out onto multiple paths on a per-packet basis to minimize hotpots (**Idea 1**). Then, if hotspot arises, MLT performs priority queueing, followed by selective dropping in case of buffer overflow, based on gradient importance (**Idea 2**). Finally, if packet drops which may potentially trigger timeout, as a final defense, MLT enables bounded loss-tolerant data transmission to avoid long retransmission delay (**Idea 3**). These 3 steps work progressively to protect MLT against the long tail latency.

4.2.1 Order-free Per-packet Load Balancing

MLT performs fast start at end-hosts, and per-packet load balancing in the network. To ensure packet independency, MLT needs careful packet construction.

Order-free packet construction. MLT first performs tensor packing before transmission (Figure 6). Gradients in a tensor are divided into separated groups, which are then packed into packets with tensor ID, layer, and offset information. When a packet reaches the receiver, gradients in the packet can be put into the corresponding addresses according to the tensor ID and gradient offset. For lost gradients, we fill in the corresponding places with zeros. For lost parameters, we use the value of the previous iteration. This ensures that the MLT receiver can still re-construct the tensor even with some lost or reordered packets.

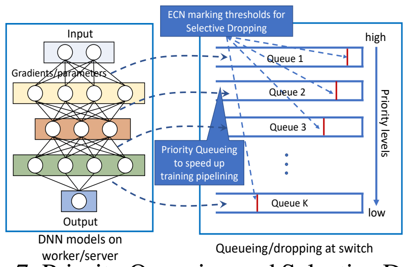

ECN marking thresholds for Selective Dropping
Input
Gradients/parameters
Output
DNN models on worker/server
Priority Queueing to speed up training pipelining
high
Queue 1
Queue 2
Queue 3
Queue K
low
Priority levels
Queueing/dropping at switch

Figure 7: Priority Queueing and Selective Dropping

Per-packet load balancing. We consider two design options for per-packet load balancing. One is to leverage the switch side per-packet ECMP [43]. The other is to give end-hosts the control of multi-path routing [46]. This can be done by source routing or label switching. In our implementation of this paper, we follow the second option as source routing delivers more predictable performance by explicitly determining the path for each packet. With the aid of per-packet load balancing, MLT effectively minimizes hotspots to a great extent.

4.2.2 Gradient-aware Packet Queueing & Dropping

Congestion may still occur even if per-packet load balancing is employed. MLT performs priority queueing and selective dropping in case of queue buildup and buffer overflow, based on the impact of gradients/packets on DNN training (§3.3).

End-host packet tagging. MLT first tags packets based on the layers and magnitudes of gradients. To decide packet priority, a straightforward solution is to map packets of each layer to a unique priority. However, this is impractical as there are usually much more DNN layers than switch priorities (typically 8 [19]). To handle it, MLT evenly distributes all layers into available priorities: it tags the packets of i-th layer with $\frac{iP}{L}$, where L and P are total numbers of DNN layers and available switch priority queues, respectively.

To encode the magnitude information, we adopt a similar approach used in gradient sparsification [66]. Given all gradients in a packet, MLT calculates the mean value and compares it against a threshold to mark whether the packet is *important* or not (indicated by ECN field in packet header §5.2)[5]. By default, the threshold is set to median of all gradients in a tensor. To minimize overhead, we sample 5% of the gradients (inspired by [66] whose sample rate is <1%).

Switch queueing and dropping. Based on the tagging information, MLT switches perform priority queueing and selective dropping as shown Figure 7. For priority queueing, it maps packets of front layers to high priority queues, which speeds up the training pipelines. For selective dropping, it decides whether to drop a packet from two dimensions. On the *layer* level, to selectively drop front layer packets, the corresponding higher priority queues are set with lower dropping thresholds (more analysis in Appendix B). On the *magnitude* level, the switch drops packets based on the importance tags; less important packets get selectively dropped. Note that there is no need for complex deep packet inspection, and both priority queueing and selective dropping can be readily implemented with commodity switches by checking DSCP and ECN fields, respectively (§5.2).

4.2.3 Bounded-loss Tolerant Data Transmission

With well-balanced traffic and deliberate queueing and dropping mechanisms, packet loss should happen rarely or, in case it happens, has limited impacts. However, in case of severe losses that may lead to timeouts, as a final resort, MLT enables a bounded-loss tolerance data transmission to avoid long retransmission latency.

Strawman design. To realize bounded-loss tolerant data transmission, a strawman solution is to let the receiver application start next iteration once it receives a certain fraction of gradients. However, this approach suffers from head-of-line (HoL) blocking due to in-order delivery nature of reliable transport protocols like TCP. Consider a 10-packet message where the second packet is lost. Though the transport layer gets 90% of data, it can only deliver the first packet up to the application as the lost second packet blocks the delivery of remaining packets. Consequently, application suffers from unnecessary retransmission delay to move forward.

The reader may wonder the feasibility of unreliable transport protocols such as UDP. While unreliable transport protocols do not have above HoL blocking problem, their best effort nature cannot guarantee the delivery of a certain portion of packets. In addition, they lack congestion control to prevent congestion collapse. Therefore, we decide to come up a new transport protocol that can guarantee the delivery of a certain fraction, say $(1-p)\%$, of packets.

Semi-reliable transmission. Before training, MLT first synchronizes among all the nodes with the following information: (1) loss-tolerant bound $p \in [0,1)$, the maximum tolerable loss fraction for a tensor; (2) ID and size of each tensor. Note that we can get this information before the training starts and tensor sizes are fixed for each of the iterations. Such information is transmitted via a reliable channel, e.g., TCP or RDMA RC (Reliable Connection) to ensure reliability.

When the correctly delivered data reaches the bound, MLT receiver sends a *stop* signal to stop the sender-side transmission. However, it is possible that the receiver may not obtain enough data after the sender transmitted all the packets for a round. To this end, after transmitting all the packets of a tensor, the sender sends a *probe* signal to the receiver to query the status. The receiver responds with a bitmap of received packets, and then the sender will re-transmit all missing packets. Such recovery process continues until the *stop* signal is received by the sender. Note that control packets such as *stop* and *probe* are transmitted over reliable channel.

Minimal rate control. In virtue of the loss-tolerance feature, MLT only requires a minimal rate control to avoid congestion

[5]Note that the magnitude of gradient is a better importance indicator than that of the original parameter. If the original parameter is small yet the gradient is large, dropping the gradient may discard update information that builds new correlation between two features from "unrelated" to "related".

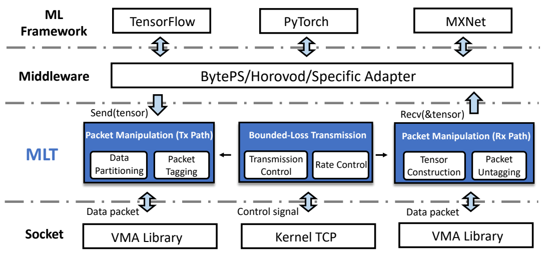


Figure 8: MLT end-host implementation overview

collapse. It uses the delay as the congestion signal and adopts a TIMELY-like [76] algorithm:

- Initially, flow starts at line rate. The sender encodes a timestamp in the data packet while the receiver periodically echoes back the timestamp via ACK. In our setting, the receiver sends one ACK for every ten arrival data packets.
- Every time the sender receives an ACK, it computes the current RTT and RTT gradient, then update the sending rate. The rule is: 1) if the current RTT is less than T_{low} or the RTT gradient is <0, it performs additive increase and sets the rate to $rate + \alpha$; 2) if the current RTT is larger than T_{high}, it performs multiplicative decrease and sets the rate to $rate \cdot (1 - \beta \cdot (1 - \frac{T_{high}}{current\_rtt}))$. In our experiment, T_{low} is 12.5μs, T_{high} is 125μs, α is 40Mbps and β is 0.8.

5 Implementation

Read only when implmenting it. Project, for example

We build a MLT prototype with Mellanox LibVMA [73] and commodity switches, and integrate it into popular ML frameworks, e.g., Tensorflow [11], PyTorch [79] and MXNet [23].

5.1 End-host Network Stack

Overview. As shown in Figure 8, MLT is implemented as a shim layer running in user space between ML framework and socket interfaces. We provide a series of universal communication interfaces that can be integrated into various ML frameworks [11, 23, 79] and distributed training middleware systems (e.g., Horovod [88] and BytePS [52]).

Universal interfaces. We design and implement basic communication primitives for common application abstractions of ML frameworks [11, 23, 52, 79]. MLT provides an asynchronous operation and completion programming model which is similar to those used in high performance communication libraries such as libverbs and libfabric. It provides two basic communication APIs `dlcp_post_send(tensor, prio_func)` and `dlcp_post_recv (tensor, loss_bound)` and an API `dlcp_poll_cq (cq)` for completion notification. The tensor abstraction is widely used by almost all popular DNN frameworks such as TensorFlow and PyTorch.

MLT sender. At the sender side, a tensor is first partitioned into several MTU-sized (excluding header overhead) segments of gradients. Then we run the priority function `prio_func` to determine the per-segment priority. We use the DSCP field in the IP header to carry the priority value. The MLT header contained in UDP payload encodes the tensor identifier, length, offset and a sequence number.

MLT receiver. At the receiver side, `dlcp_post_recv` takes the tensor received and the loss-tolerant bound as input. Before data transmission, the sender and receiver do a rendezvous to allocate receiving buffer in advance. Upon receiving a new packet, the receiver copies its gradients to the pre-allocated memory according to its offset.

Data & signal transmission. We implement MLT using both UDP and TCP. Inspired by [40], we separate data transfers and control signals, and only provide full reliability for control signals (flow start/finish, retransmisison request, stop request/confirm) whose traffic size is much smaller. To ensure reliability, we use TCP to carry control signals. To minimize the loss of control signals, we reserve a separate priority for them at the switch. We find that control packets are rarely dropped in practice. To achieve high throughput, we adopt UDP in Mellanox LibVMA [73] (instead of Linux kernel), a high-performance user space network stack.

Retransmission. MLT implements selective acknowledgement (SACK) to manage retransmission. The buffer of a transmitting tensor is shared by both the application and MLT library until the corresponding completion is generated. Thus, there is no need to maintain an additional buffer for retransmission of unacknowledged packets.

RDMA Implementation feasibility. Today, RDMA is widely used to accelerate distributed ML training [52, 106]. The reader may wonder how to implement MLT in RDMA NIC. However, we did a feasibility study and noticed that today's RDMA NIC hardware still cannot provide enough programmability to implement complex transport functionalities efficiently. For example, even on the state-of-the-art NVIDIA ConnectX-7 NIC [8] which supports programmable congestion control, users still cannot modify packet retransmission logic inside the NIC. If we onload MLT's complex functionalities, e.g., semi-reliable transmission, to user space like previous work [53], we may lose the real benefits of hardware offloading. Therefore, we leave a full hardware implementation of MLT as future work.

5.2 Switch Configurations

We implement priority queueing and selective dropping using built-in functions of commodity switches.

Priority queueing (with DSCP). We enable strict priority queueing and classify packets into the corresponding priority queues based on the DSCP field [19, 22].

Selective dropping (with ECN). Current switching chips cannot *push out* packets that are already stored in the switch buffers. Therefore, we can only selectively drop packets at the ingress. To this end, inspired by Aeolus [45], we use RED/ECN function [18], which is supported by commodity switches. In current implementations, when the switch queue size exceeds the ECN marking threshold, the switch will mark the arrival ECN-capable packets and *drop* non-ECN-capable packets. Hence, to implement selective dropping, we tag the packets carrying large gradients with ECN-capable at the sender side. To implement layer-wise selective dropping, e.g., packets from the front layers are easier to drop than that from the back layers, we set smaller ECN marking thresholds for higher priority queues.

5.3 ML Framework Integration

MLT can be directly integrated with ML frameworks such as TensorFlow [11], PyTorch [79], and MXNet [23] or indirectly integrated with some distributed training middleware systems such as Horovod [88] and BytePS [52]. Typically, ML frameworks have their own distributed training implementations. They tend to choose a specific RPC or messaging library and build an abstraction over it. For example, MXNet uses PS-Lite [60] and builds a key-value store over it, while PyTorch prefers collective communication API and can have multiple backends such as Gloo [32], MPI [28] or NCCL [5]. These communication abstraction layers decide which nodes are communicating with each other in one iteration and are usually built on the top of point-to-point communication primitives. To directly integrate MLT with ML frameworks, we only need to re-implement their communication abstraction layers using MLT's interfaces. To indirectly integrate MLT with distributed training middleware systems, we have to replace the point-to-point communication with MLT, and construct the All-reduce scheme or PS topology when necessary. As BytePS supports TensorFlow, PyTorch and MXNet, in our prototype, we integrate MLT into BytePS [52].

6 Evaluation

We evaluate MLT with a combination of testbed experiments and larger-scale simulations. The highlights include:

- In testbed experiments, we evaluate MLT across different DNN models, ML frameworks and synchronization paradigms (§6.1). Compared to the state-of-the-art work BytePS [52] or PyTorch FSDP [115], MLT achieves up to 62.2% (PS), 10.2% (Ring) and 35.5% (FSDP, for LLMs) training speedup without hurting the accuracy.
- In large-scale simulations, we compare MLT against various advanced datacenter transport protocols with realistic DNN training traffic (§6.2). MLT achieves 43.1%/91.8% lower average/tail FCTs over DCTCP [14] and 6.7%/10.3% lower average/tail FCTs over NDP [38] under ResNet50-induced traffic, without modifying switch hardware.
- Deep-dive into MLT design (§6.3) shows that each of its design components contributes effectively to its performance. We further quantify the impacts of loss-tolerant bounds and microsecond-level RTO_{min}.

6.1 Testbed Experiments

6.1.1 Experimental Setup

Testbed: Our testbed (Figure 9) has 8 physical GPU servers, each with 8 RTX3090 GPUs, 80 CPU cores (Intel Xeon Gold 5218R), 256GB memory, 2 Mellanox ConnectX5 100Gbps NICs, and 4 Mellanox SN2100 switches running

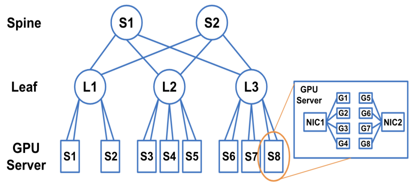


Figure 9: Testbed Topology

Onyx 3.7.1134 OS. It forms a leaf-spine topology with one spine switch and three leaf switches in physical. Each leaf switch has two 100Gbps links connecting to the spine switch, thus logically we have two spine switches. The three racks contain 2, 3, 3 GPU servers, respectively. Each server has two 100Gbps links connecting to the leaf switch. We further divide one physical servers into 8 docker containers, each with 1x GPU, 10x CPU cores, 32GB memory and 25Gbps virtual NIC[6]. Ultimately, we get a 64-node testbed with 2:1 to 3:1 over-subscription, a normal ratio in production [92].

Models and datasets: We use four models and two datasets in our experiments. Our models include three image classification tasks: VGG16 [91], ResNet50 [41] and GoogleNet [93] training on the synthetic data with the same image size as ImageNet [29], and one translation task: Transformer [98] training on SQuAD [85]. We run experiments on three ML frameworks: TensorFlow, PyTorch, MXNet with two parameter synchronization paradigms: PS (colocated and #servers = #workers) and Ring All-reduce.

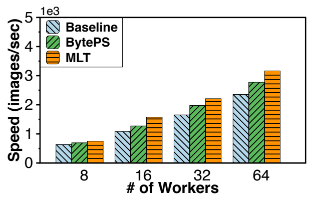
Baselines and metrics: We mainly compare MLT with the vanilla ML frameworks (baseline) and BytePS [52] with cross global barrier enabled. Basically, BytePS incorporates tensor partition and priority scheduling of ByteScheduler [81] and has better code robustness, thus representing a state-of-the-art in communication optimization of DNN training. For image classification models, we use the # of images processed per second as the speed metric, and for transformer we measure the # of questions processed per second [85].

Parameter settings: The batch sizes of VGG16, ResNet50, GoogleNet and Transformer are 32, 32, 32, and 10 samples (images or questions) per GPU, referring to the settings in [81]. Switches have 4MB shared memory, and 8 queues per port. We use DCTCP [14] as the transport protocol for baseline and BytePS. To ensure a fair comparison between MLT and BytePS, we open the multiple connections function [2] of BytePS, i.e., two connections per 25Gbps, to make sure it can saturate the bandwidth. RTO_{min} is $1ms$ and initial window size is 20. With selective dropping, in our experiments, we conservatively set loss-tolerant bound to 10% for MLT (see §6.3 for deep-dive on impact of loss-tolerant bounds).

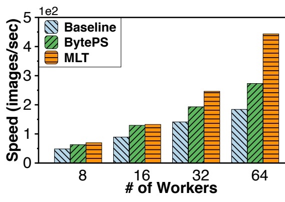
6.1.2 Results

Overall, across different DNN models, ML frameworks and sync paradigms, MLT achieves remarkable training speedup over state-of-the-art. Our measurement also shows that the

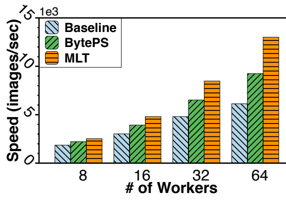
[6]We use SR-IOV to separate the physical NIC. SR-IOV can achieve nearly the same performance as the non-virtualized environments [30].



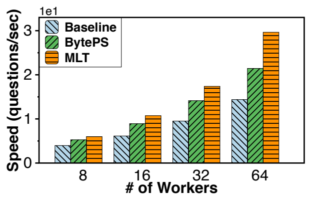
(a) ResNet50



(b) VGG16

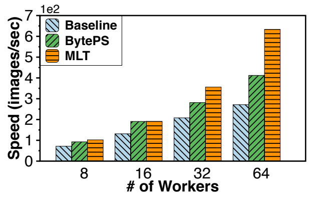


(c) GoogleNet

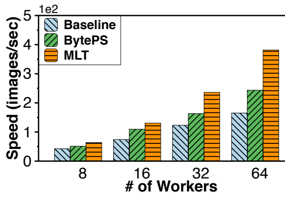


(d) Transformer

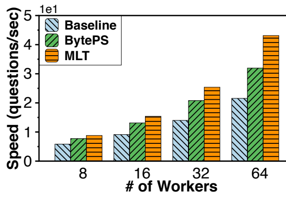
Figure 10: Speedup under different DNN models using TensorFlow under PS (all converge with the same epochs).



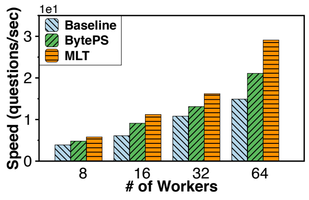
(a) MXNet, VGG16



(b) PyTorch, VGG16

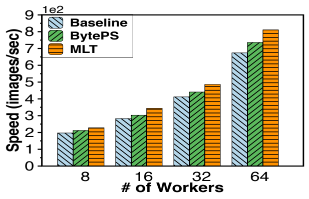


(c) MXNet, Transformer

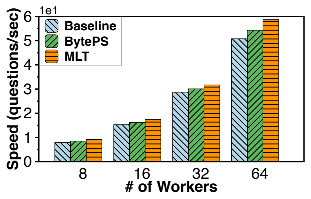


(d) PyTorch, Transformer

Figure 11: Speedup under other frameworks: MXNet and PyTorch (results with ResNet50 and GoogleNet in Appendix).

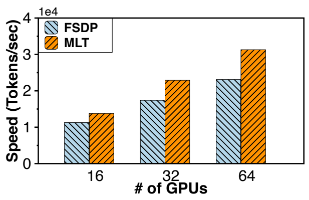


(a) VGG16

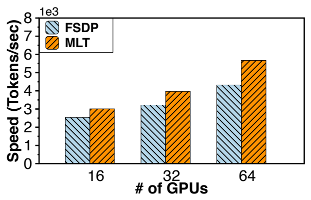


(b) Transformer

Figure 12: Speedup under Ring All-reduce.



(a) T5-3B



(b) T5-11B

Figure 13: Speedup under Large Language Models.

additional CPU cost introduced by MLT is <1%. Note that our results below are all based on the condition that *the models converge to the same accuracy with the same iterations.*

Speedup under different DNN models: Figure 10 shows performance of MLT over baseline and BytePS on four DNN models using TensorFlow under PS. From the figure, we make the following three observations. First, MLT performs the best of all on all models. Specifically, MLT outperforms BytePS by 14.1%–62.2% and baseline by 34.5%–141%, across the four models. The main reason is that the default reliable transport is sensitive to packet losses, which may trigger timeouts and cause millisecond-level delay. During the training, we observed around 0.15% packet losses from the buffer counting function provided by our switch [3]. Second, the improvement of MLT becomes more significant as the # of workers increases. As we can see, MLT outperforms BytePS by 9.34% to 10.7% when the # of workers is 8, and by 14.1% to 62.2% when it increases to 64. This is expected because as the network pressure grows higher, packet losses become more frequent. Third, MLT achieves more speedup in VGG16 than other models. As shown in Figure 10(a), MLT achieves up to 62.2% speedup in VGG16, more than that in other models. The reason is that VGG16 is communication-bound and it has the largest communication-to-computation ratio.

Speedup under different ML frameworks: Figure 11 shows the performance of MLT over the baseline and BytePS in PyTorch and MXNet by training VGG16 and Transformer under PS (ResNet50 and GoogleNet in Appendix-C, Figure 22). Note that the native PyTorch Distributed Data Parallel does not support PS, so we use the modified version [6] as the baseline for PyTorch. We observe similar trends as above in TensorFlow. Overall, we find that MLT outperforms BytePS by 12.0%–53.6% and 15.3%–56.6% in MXNet and PyTorch respectively, and they both achieve the best speedup with VGG16 due to the same reason explained above. This experiment shows that MLT can deliver persistent performance improvement across different ML frameworks.

Speedup under Ring All-reduce: The above experiments all use PS, which is widely used in both academia and industry [50, 52, 60, 81], especially for the large and sparse models [61]. Here, we further evaluate the performance of MLT under Ring All-reduce. Please note that BytePS [52] by default does not support the Ring All-reduce communication, we adapt it to support Ring. Figure 12 shows the results of VGG16 and Transformer (ResNet50 and GoogleNet in Appendix-C, Figure 23). In all cases, we see clear performance improvement with MLT, though not significant. Compared to BytePS, MLT achieves 6.89%–10.2% improvement; in comparison, BytePS achieves 5.97%–8.21% improvement over the baseline. As expected, we observe less speedup in Ring All-reduce than that in PS, and the key reason is that: packet loss is rare in Ring All-reduce, thus it is immune from long tail latency caused by timeout. Nevertheless, MLT still delivers better performance over BytePS due to the fine-grained per-packet load balancing and reduction of data volume in transmission with bounded loss tolerance.

Speedup under Large Language Models (LLMs): Recently, LLMs like ChatGPT [10] have gained popularity and received considerable attention from the community. To support training or fine-tuning of LLMs, a popular approach is to divide the parameters, gradients, and optimizer states equally into each GPU's memory during the data parallelism. Before computing a layer of a model, an all-reduce communication is performed to ensure each GPU possesses a full copy of the current layer. This approach reduces GPU memory footprint through frequent communication. Notable implementations of this approach include Zero Redundancy Optimizer (ZeRO) [84] and Fully Sharded Data Parallel (FSDP) [115].

To evaluate MLT under LLMs, we replace the inter-server communication module of the PyTorch FSDP [115] (using NCCL [5]) with MLT. For the communication of optimizer states that are used for gradient calculation, i.e., gradient momentums and historical gradient values, we treat them the same as gradients. Considering that each GPU does not store the full copy of parameters, we cannot use the past parameters to replace the missing new parameters. To handle this issue, we mark all parameter packets as important packets. This does not significantly alter the proportion of important packets, as the optimizer states dominate the traffic volume in FSDP [84, 115]. Given the limitation of compute power of our testbed, we chose the HuggingFace T5 model [83], a popular transformer-based open-source LLM, 3/11-billion versions and fine-tune them on the WikiHow dataset [55] for the text summarization task.

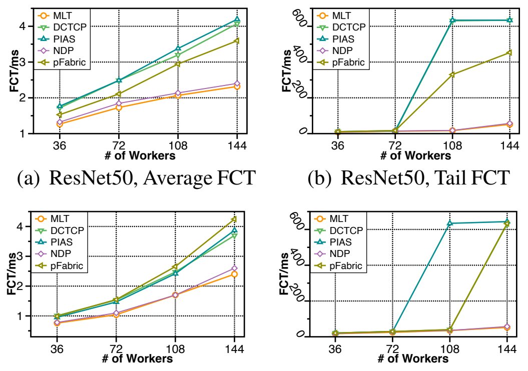
Figure 13 shows the results. In our experiment, we find that for the T5-11B model, 8 RTX3090s are insufficient to accommodate it, so we show the results with 16, 32 and 64 GPUs respectively. Compared to the PyTorch FSDP, MLT achieves an improvement of 22.1%-35.5% for T5-3B and 18.5%-31.2% for T5-11B. We observe more speedup compared to Figure 12 with the same Ring All-reduce setting, this is because FSDP involves more frequent communication and a larger volume of data transfer for LLMs. Moreover, we monitor the model loss changes during the fine-tuning process and find that MLT does not affect the model convergence.

6.2 Large-scale Simulations

6.2.1 Simulation Setup

Topology and traffic: As [15, 19], we choose a leaf-spine topology with 4 core, 9 ToR switches and 144 hosts. Each ToR switch is connected to 16 hosts using 100Gbps links and 4 core switches using 4×100Gbps links. The base RTT between two servers (4 hops) is $24\mu s$. Each switch port has 512KB buffer. For network traffic, we use the realistic workloads derived from training ResNet50 and GoogleNet under PS. We evenly distribute workers and parameter servers across all the racks, with the ratio of 3:1. We also obtain the computation time and tensor sizes from our testbed.

Schemes compared: We use DCTCP [14] as the baseline, as it is widely used in production. We also compare MLT with PIAS [19], pFabric [15], NDP [38] (simulated on htsim). TCP initial window is 10, and ECN marking threshold is 65 packets [14]. RTO_{min} is set to 10ms by default [20, 97] (We also ran simulations with 5ms and 1ms RTO_{min}, and observed similar trends). DupACKs is 3 for DCTCP and PIAS, and DelayAck is disabled. DCTCP and PIAS use per-flow ECMP while pFabric uses per-packet ECMP.



(a) ResNet50, Average FCT (b) ResNet50, Tail FCT
(c) GoogleNet, Average FCT (d) GoogleNet, Tail FCT
Figure 14: Large-scale simulations (network metrics).

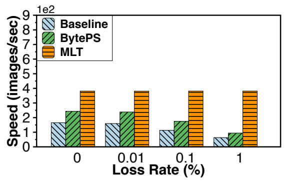
6.2.2 Results

Figure 14 shows the average and tail FCTs of MLT versus other schemes for ResNet50 and GoogleNet induced traffic at varying network scales. In general, MLT delivers the best performance. For ResNet50, MLT achieves up to 43.1%, 44.8%, 35.5%, 6.7% lower average FCT and 91.8%, 91.8%, 88.6%, 10.3% lower tail FCT compared to DCTCP, PIAS, pFabric and NDP, respectively. For GoogleNet, MLT delivers up to 35.1%, 38.0%, 43.5%, 7.7% lower average FCT and 91.7%, 91.9%, 91.7%, 8.7% lower tail FCT over these schemes, respectively. From the above results, we make the following observations:

- *MLT preforms the best in all settings.* MLT achieves the best performance in all workloads and network scales. The main reason is that all the other algorithms suffer from packet loss and retransmission to ensure reliability, while MLT tolerates certain packet loss.
- *MLT significantly reduces the tail FCT.* Relative to average FCT, MLT reduces the tail more significantly. The reason is that retransmission timeout greatly increases the tail of other algorithms, while MLT tolerates packet loss and is free of retransmission timeout. We also note that NDP can achieve similar performance as MLT. However, it requires special switch hardware and is not readily deployable.
- *The speedup of MLT is more notable as the network scale increases.* In general, MLT reduces FCT more significantly at larger scale. The reason is that, as the scale increases, prior solutions experience more packet loss, thus performance degradation, whereas MLT is immune to overhead by packet loss with the bounded-loss tolerance.

6.3 Deep Dive

MLT under gray failure of links [testbed]: Fault-tolerance and reliability are crucial for distributed training, especially for large models. We find that the loss tolerance feature of MLT can be a good solution in the gray failure of network links, i.e., hard-to-detect but persistent link random packet loss, which may be caused by subtle hardware malfunctions, firmware bugs, or environmental interference [48]. A measurement work in Microsoft categorizes a network problem when the loss rate is larger the 0.1% [37]. Here we evaluate the performance of


(a) VGG16

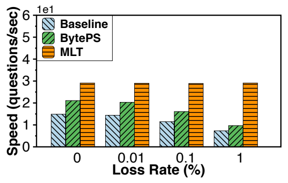

(b) Transformer

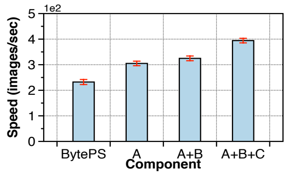
Figure 15: Speedup under gray failure of links.

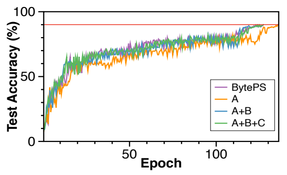
MLT under gray failure on our 64 GPUs testbed. We assume that a worker-to-switch link has a gray failure, and emulate packet loss by adding random packet dropping to the mapped NICs by using the tc qdisc tool in linux. The loss rate is set to 0, 0.01%, 0.1% and 1% in the experiment and we use PyTorch as the framework. Figure 15 shows the results of VGG16 and Transformer. Figure 21 shows ResNet50 and GoogleNet in Appendix-C. We can see that the baseline and BytePS start to suffer from noticeable performance degradation (12.2%-31.5% and 15%-28.3%) when the loss rate is 0.1% and the degradation is significant (25.7%-61.7% and 31.2%-61.3%) when the loss rate is 1%. As a comparison, MLT maintains its performance in all the loss scenarios.

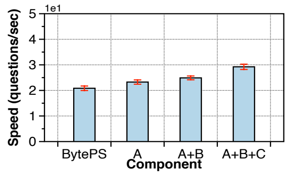
Design component effectiveness [testbed]: MLT consists of three main components: A) bounded-loss tolerant data transmission, B) gradient-aware queueing and dropping, and C) order-free per-packet load balancing. We now look into the effectiveness of each component. To do so, we set BytePS [52] as the baseline, and start MLT with component A and then gradually add B and C one by one (setting I). Then, we start with component C (using UDP with the minimal rate control of MLT) and gradually add B and A (setting II). We measure the speeds of training ResNet50, Transformer, VGG16 and GoogleNet under PyTorch with 64 GPUs, and we repeat the experiment 10 times and compute the mean and standard error. Correspondingly, we also record the convergence curves (epoch-to-accuracy, ETA). The results of VGG16 and Transformer are shown in Figure 16 for setting I and Figure 17 for setting setting II (ResNet50 and GoogleNet in Appendix-C, Figure 24 and Figure 25). For CNN models, we use top-1 accuracy; for Transformer, we use exact match (EM) as the test accuracy metric.

From the results, we see that each component contributes effectively to the overall performance. Specifically, in setting I, compared to BytePS, MLT with component A only can improve the training speed (left figures). While it requires a bit more rounds to converge to the same accuracy (right figures, due to the packet loss as we set the loss bound as 10%), we find that the end-to-end training time still improves (decided by both unit speed and # of epochs). Then, with component B added, the ETA of MLT is significantly improved (almost as good as BytePS) as the training speed increases. The main reason is due to selective dropping which preserves important gradients from being dropped. Finally, after incorporating component C, MLT maintains its good ETA while improving remarkably in training speed. This is because of the better network utilization brought by perfect load balancing.

In setting II, spreading gradients out at packet granularity (C)


(a) VGG16, Speed


(b) VGG16, ETA


(c) Transformer, Speed

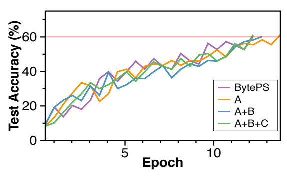
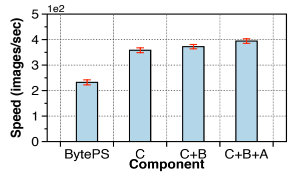
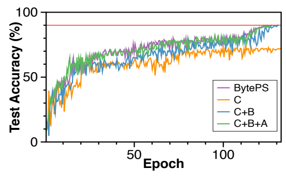
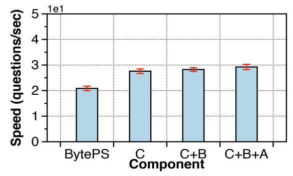

(d) Transformer, ETA

Figure 16: Effectiveness of design components (I).


(a) VGG16, Speed


(b) VGG16, ETA


(c) Transformer, Speed

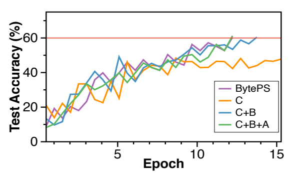
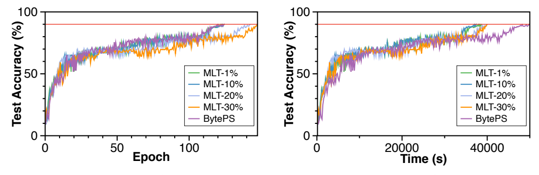

(d) Transformer, ETA

Figure 17: Effectiveness of design components (II).

can effectively improve the training speed, but requiring more epochs to converge and may reduce the accuracy. After adding gradient-aware packet queueing and dropping (B), the speed is sightly increased and convergence is significantly improved, only needs a few more epochs than the baseline. The reason is that priority queueing can speed up the communication-computation pipeline and selective dropping can protect more important gradients. Finally, with the bounded-loss tolerance transmission (A), it can further speedup the training and achieving the same convergence speed and accuracy as the baseline.

Impact of loss-tolerant bounds [testbed]: We inspect the convergence behaviors of MLT with different loss-tolerant bounds by training VGG16 and Transformer in Figure 18 (ResNet50 and GoogleNet in Appendix-C, Figure 26). We set the bound as 1%, 10%, 20% and 30%, respectively. Figure 18(a)(c) show curves of ETA. We can see that with 1% and 10% loss-tolerant bounds, the curves are all almost in line with BytePS; with 20% and 30% loss-tolerant bounds, it requires more epochs to converge to the same accuracy. We also measured the bounds of other widely-used models with MLT in Table 5 of Appendix-C. The results indicate that, with the optimization of gradient-aware dropping, MLT can tolerate more packet loss than pure random loss shown in Table 1. This motivates us to set the loss-tolerant bound as 10% in our testbed experiment (§6.1). Figure 18(b)(d) show results of time-to-accuracy (TTA). Compared to BytePS, MLT converges faster under all


(a) VGG16, ETA (b) VGG16, TTA

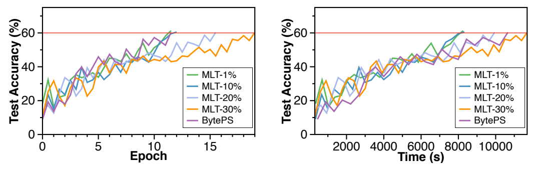

(c) Transformer, ETA (d) Transformer, TTA

Figure 18: Impact of loss-tolerant bounds.

loss-tolerant bounds, except for MLT-30% under Transformer. Meanwhile, we find that 1% and 10% loss-tolerant bounds take almost the same time to converge. This implies that we might not need to struggle with fine-tuning the loss-tolerant bound in order to achieve satisfactory performance. However, the relationship between the model architecture and its loss-tolerant bounds remains an open question for future study.

Impact of microsecond RTO$_{min}$ [simulation]: As a complement to testbed, we measure the tail FCT when RTO_{min} is $<1ms$ using simulation. Specifically, we choose three RTO_{min}s: $25\mu s$ which is close to our $24\mu s$ base RTT, and two intermediate values: $100\mu s$ and $500\mu s$. We conduct the simulations with GoogleNet-induced traffic under different compression ratios, and show the results in Figure 19. From the figure, we make two observations. First, while the RTO_{min} is sub-ms level, the tail FCT is still around $10ms$ or above (MLT can reduce it to $4.12ms$). This is due to the same reason as explained in §2.2—tail FCT is mainly decided by large actual RTO amplified by consecutive packet losses and spurious retransmissions. For example, we observe 4.2KB spurious retransmission from the tail flow (the size is 127.4KB) and 0.97% packet loss under $25\mu s$ RTO_{min} and $32\times$ compression. MLT can reduce the tail to $4.12ms$. Second, the curves turn to flat after a certain compression ratio, which implies that further compressing the gradient volume does not help to reduce the tail FCT. This is also in line with our observation in §2.2. As a final remark, while we showcase the μs-level RTO_{min} here, we remind readers that in production RTO_{min} is typically set at ms-level.

7 Open Questions and Discussion

MLT is, by no means, a *full stop* to ML-specific transport. We have left a series of open questions from theoretical aspects of ML characteristics to practical implementation/deployment throughout the paper. Here, we discuss a few more. Due to the space limit, we move the discussion of "co-existing with non-DNN traffic", "flexible selective dropping" and "MLT vs SRD" to the Appendix-D.

MLT for other parallelism schemes. While MLT is mainly designed for data parallelism, its core ideas may provide insights for designing network transport for other schemes, i.e., model/pipeline/tensor parallelism [51]. Some MLT mechanisms can be directly used. For example, the order-free

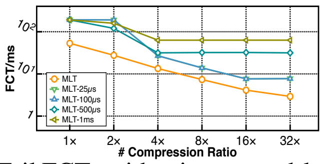


Figure 19: Tail FCTs with microsecond-level RTO_{min}.

pre-packet load balancing remains effective for activations and propagated gradients in model parallelism. Others may need to be modified to adapt to the nature of model parallelism. For example, packets in transmission for model parallelism are primarily used for FP/BP computations, which makes their priorities/loss-tolerance properties different from the model synchronization/update phases in data parallelism. We leave the theoretical analysis of this for future study.

Multi-tenant cloud environment. In multi-tenant cloud environments, e.g., EC2 and Azure, we will not be able to manipulate the underlying network protocols and switches. For example, the traffic may be encapsulated inside a tunnel (like VXLAN) and switches cannot access those fields to perform priority queueing or dropping. Furthermore, the cloud providers typically do not allow tenants to access the queues of the underlying network for security reasons. Thus, we acknowledge that MLT does not work for such scenarios.

8 Related Work

Besides the closely related works discussed in §2.2, there exist some other solutions to improve the communication of DNN training. For example, RDMA [17, 67] and NCCL [5] provide higher bandwidth to speedup tensor transmission. BlueConnect [27] and PLink [70] design novel communication patterns with network topology awareness for gradient synchronization at each iteration for better performance and robustness. GPipe [49] and PipeDream [78] overlap communication with computation in model parallelism. More recently, SwitchML [87] and ATP [58] leverages in-network aggregation to reduce the communication overhead in the network. Note that these works are orthogonal to MLT.

9 Conclusion

This paper presented MLT, a domain-specific network transport exploiting the special properties of machine learning to optimize distributed DNN training. MLT consists of three key ideas: 1) order-free per-packet load balancing, 2) gradient-aware packet queueing and dropping, and 3) bounded-loss tolerant data transmission. Extensive testbed experiments and simulations have shown the promise of MLT.

Acknowledgement

We thank the anonymous NSDI reviewers and our shepherd Costin Raiciu for their feedback and suggestions. This work is supported in part by Hong Kong RGC TRS T41-603/20R, GRF 16213621, ITF ACCESS, NSFC 62062005, Key-Area Research and Development Program of Guangdong Province (2021B0101400001), and the Turing AI Computing Cloud (TACC) [105]. Kai Chen is the corresponding author.

References

[1] Ieee. 802.11qbb. priority based flow control https://1.ieee802.org/dcb/802-1qbb/, 2011.

[2] Byteps best practice: https://github.com/bytedance/byteps/blob/master/docs/best-practice.md?plain=1#L26, 2020.

[3] Mellonax switch: https://www.mellanox.com/products/ethernet-switches, 2020.

[4] Mlperf training results resnet50: https://mlperf.org/training-results-0-6, 2020.

[5] Nvidia collective communications library: https://www.nvidia.com/en-us/data-center/nvlink/, 2020.

[6] Pytorch distributed data parallel supporting ps: https://github.com/bytedance/byteps/blob/master/docs/DistributedDataParallel.md, 2020.

[7] Imagenet-100: https://www.kaggle.com/datasets/ambityga/imagenet100, 2022.

[8] Connectx-7: https://nvdam.widen.net/s/csf8rmnqwl/infiniband-ethernet-datasheet-connectx-7-ds-nv-us-2544471, 2023.

[9] Nvidia h100 tensor core gpu: https://www.nvidia.com/en-us/data-center/h100/, 2023.

[10] Openai chatgpt: https://chat.openai.com, 2023.

[11] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, 2016.

[12] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In *NIPS*, 2017.

[13] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, et al. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 503–514, 2014.

[14] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *SIGCOMM*, 2010.

[15] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. In *SIGCOMM*. ACM, 2013.

[16] Arnold O Allen. *Probability, statistics, and queueing theory*. Academic press, 2014.

[17] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, et al. Empowering azure storage with rdma. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 49–67, 2023.

[18] Wei Bai, Kai Chen, Li Chen, Changhoon Kim, and Haitao Wu. Enabling ecn over generic packet scheduling. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 191–204. ACM, 2016.

[19] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 455–468, 2015.

[20] Wei Bai, Shuihai Hu, Kai Chen, Kun Tan, and Yongqiang Xiong. One more config is enough: Saving (dc) tcp for high-speed extremely shallow-buffered datacenters. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 2007–2016. IEEE, 2020.

[21] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *COMPSTAT'2010*. Springer, 2010.

[22] Li Chen, Kai Chen, Wei Bai, and Mohammad Alizadeh. Scheduling mix-flows in commodity datacenters with karuna. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 174–187, 2016.

[23] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[24] Yanpei Chen, Rean Griffith, Junda Liu, Randy H Katz, and Anthony D Joseph. Understanding tcp incast throughput collapse in datacenter networks. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, pages 73–82, 2009.

[25] Peng Cheng, Fengyuan Ren, Ran Shu, and Chuang Lin. Catch the whole lot in an action: Rapid precise packet loss notification in data center. In *11th*

{*USENIX*} *Symposium on Networked Systems Design and Implementation* ({*NSDI*} *14)*, pages 17–28, 2014.

[26] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder–decoder for statistical machine translation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014.

[27] Minsik Cho, Ulrich Finkler, and David Kung. Blueconnect: Novel hierarchical all-reduce on multi-tired network for deep learning. In *Proceedings of the Conference on Systems and Machine Learning (SysML)*, 2019.

[28] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. The mpi message passing interface standard. In *Programming environments for massively parallel distributed systems*, pages 213–218. Springer, 1994.

[29] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[30] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. High performance network virtualization with sr-iov. *Journal of Parallel and Distributed Computing*, 72(11):1471–1480, 2012.

[31] Nikoli Dryden, Naoya Maruyama, Tim Moon, Tom Benson, Andy Yoo, Marc Snir, and Brian Van Essen. Aluminum: An asynchronous, gpu-aware communication library optimized for large-scale training of deep neural networks on hpc systems. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2018.

[32] Facebook. Gloo: Collective communications library with various primitives for multi-machine training., 2020.

[33] Jiawei Fei, Chen-Yu Ho, Atal Narayan Sahu, Marco Canini, and Amedeo Sapio. Efficient sparse collective communication and its application to accelerate distributed deep learning. In *SIGCOMM*, 2021.

[34] Li Fei-Fei, Rob Fergus, and Pietro Perona. Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories. *Computer Vision and Pattern Recognition Workshop*, 2004.

[35] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[36] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 202–215. ACM, 2016.

[37] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 139–152, 2015.

[38] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 29–42, 2017.

[39] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H Campbell. Tictac: Accelerating distributed deep learning with communication scheduling. *arXiv preprint arXiv:1803.03288*, 2018.

[40] E. He, J. Leigh, O. Yu, and T. A. Defanti. Reliable blast udp : predictable high performance bulk data transfer. In *Proceedings. IEEE International Conference on Cluster Computing*, pages 317–324, 2002.

[41] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

[42] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[43] Christian Hopps et al. Analysis of an equal-cost multi-path algorithm. Technical report, RFC 2992, November, 2000.

[44] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R Ganger, Phillip B Gibbons, and Onur Mutlu. Gaia: Geo-distributed machine learning approaching lan speeds. In *NSDI*, 2017.

[45] Shuihai Hu, Wei Bai, Gaoxiong Zeng, Zilong Wang, Baochen Qiao, Kai Chen, Kun Tan, and Yi Wang. Aeolus: A building block for proactive transport in datacenters. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures,*

and protocols for computer communication, pages 422–434, 2020.

[46] Shuihai Hu, Kai Chen, Haitao Wu, Wei Bai, Chang Lan, Hao Wang, Hongze Zhao, and Chuanxiong Guo. Explicit path control in commodity data centers: Design and applications. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 15–28, 2015.

[47] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[48] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 150–155, 2017.

[49] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*, pages 103–112, 2019.

[50] Anand Jayarajan, Jinliang Wei, Garth A. Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based parameter propagation for distributed dnn training. In *Proceedings of Systems and Machine Learning (SysML)*, 2019.

[51] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems*, 1:1–13, 2019.

[52] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed {DNN} training in heterogeneous gpu/cpu clusters. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 463–479, 2020.

[53] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter rpcs can be general and fast. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 1–16, 2019.

[54] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[55] Mahnaz Koupaee and William Yang Wang. Wikihow: A large scale text summarization dataset. *arXiv preprint arXiv:1810.09305*, 2018.

[56] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The cifar-10 dataset, 2014.

[57] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[58] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael M Swift. Atp: In-network aggregation for multi-tenant learning. In *NSDI*, pages 741–761, 2021.

[59] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[60] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *OSDI*, 2014.

[61] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. *Advances in Neural Information Processing Systems*, 27:19–27, 2014.

[62] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J Smola. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 661–670, 2014.

[63] Y. Li, J. Park, M. Alian, Y. Yuan, Z. Qu, P. Pan, R. Wang, A. Schwing, H. Esmaeilzadeh, and N. S. Kim. A network-centric hardware/algorithm co-design to accelerate distributed training of deep neural networks. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 175–188, 2018.

[64] Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. Asynchronous parallel stochastic gradient for nonconvex optimization. In *Advances in Neural Information Processing Systems*, pages 2737–2745, 2015.

[65] Xiangru Lian, Ce Zhang, Huan Zhang, Cho-Jui Hsieh, Wei Zhang, and Ji Liu. Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 5330–5340, 2017.

[66] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*, 2017.

[67] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K Panda. High performance rdma-based mpi implementation over infiniband. *International Journal of Parallel Programming*, 32(3):167–198, 2004.

[68] Yuanwei Lu, Guo Chen, Larry Luo, Kun Tan, Yongqiang Xiong, Xiaoliang Wang, and Enhong Chen. One more queue is enough: Minimizing flow completion time with explicit priority notification. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[69] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. Parameter hub: a rack-scale parameter server for distributed deep neural network training. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 41–54, 2018.

[70] Liang Luo, Peter West, Jacob Nelson, Arvind Krishnamurthy, and Luis Ceze. Plink: Efficient cloud-based training with topology-aware dynamic hierarchical aggregation. In *Proceedings of the 3rd MLSys Conference*, 2020.

[71] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European conference on computer vision (ECCV)*, pages 116–131, 2018.

[72] Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks. *arXiv preprint arXiv:1804.07612*, 2018.

[73] Mellanox. Messaging accelerator (vma), 2019.

[74] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. Regularizing and optimizing lstm language models. *arXiv preprint arXiv:1708.02182*, 2017.

[75] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

[76] Radhika Mittal, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, David Zats, et al. Timely: Rtt-based congestion control for the datacenter. In *SIGCOMM'15*. ACM, 2015.

[77] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 221–235, 2018.

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

[79] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[80] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009.

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

[82] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 1999.

[83] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020.

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

[85] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text. *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, 2016.

[86] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, 2011.

[87] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. Scaling distributed machine learning with in-network aggregation. *arXiv preprint arXiv:1903.06701*, 2019.

[88] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[89] Leah Shalev, Hani Ayoub, Nafea Bshara, and Erez Sabbag. A cloud-optimized transport protocol for elastic and scalable hpc. *IEEE Micro*, 40(6):67–73, 2020.

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

[91] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[92] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. *ACM SIGCOMM computer communication review*, 45(4):183–197, 2015.

[93] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[94] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114. PMLR, 2019.

[95] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[96] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. Let it flow: Resilient asymmetric load balancing with flowlet switching. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 407–420, 2017.

[97] Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G Andersen, Gregory R Ganger, Garth A Gibson, and Brian Mueller. Safe and effective fine-grained tcp retransmissions for datacenter communication. *ACM SIGCOMM computer communication review*, 39(4):303–314, 2009.

[98] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[99] Subhashini Venugopalan, Marcus Rohrbach, Jeffrey Donahue, Raymond Mooney, Trevor Darrell, and Kate Saenko. Sequence to sequence-video to text. In *Proceedings of the IEEE international conference on computer vision*, pages 4534–4542, 2015.

[100] Xinchen Wan, Hong Zhang, Hao Wang, Shuihai Hu, Junxue Zhang, and Kai Chen. Rat - resilient allreduce tree for distributed machine learning. In *4th Asia-Pacific Workshop on Networking*, APNet '20, page 52–57, New York, NY, USA, 2020. Association for Computing Machinery.

[101] Jianqiao Wangni, Jialei Wang, Ji Liu, and Tong Zhang. Gradient sparsification for communication-efficient distributed optimization. In *NIPS*, 2018.

[102] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[103] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 595–610, 2018.

[104] Eric P Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. Petuum: A new platform for distributed machine learning on big data. *IEEE Transactions on Big Data*, 2015.

[105] Kaiqiang Xu, Xinchen Wan, Hao Wang, Zhenghang Ren, Xudong Liao, Decang Sun, Chaoliang Zeng, and Kai Chen. Tacc: A full-stack cloud computing infrastructure for machine learning tasks. *arXiv preprint arXiv:2110.01556*, 2021.

[106] Jilong Xue, Youshan Miao, Cheng Chen, Ming Wu, Lintao Zhang, and Lidong Zhou. Fast distributed deep learning over rdma. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–14, 2019.

[107] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Advances in neural information processing systems*, pages 3320–3328, 2014.

[108] Chen Yu, Hanlin Tang, Cedric Renggli, Simon Kassing, Ankit Singla, Dan Alistarh, Ce Zhang, and Ji Liu. Distributed learning over unreliable networks. In *International Conference on Machine Learning*, pages 7202–7212. PMLR, 2019.

[109] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

[110] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. Detail: reducing the flow completion time tail in datacenter networks. In *SIGCOMM*. ACM, 2012.

[111] Chiyuan Zhang, Samy Bengio, and Yoram Singer. Are all layers created equal? *arXiv preprint arXiv:1902.01996*, 2019.

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

[113] Hong Zhang, Junxue Zhang, Wei Bai, Kai Chen, and Mosharaf Chowdhury. Resilient datacenter load balancing in the wild. In *SIGCOMM*. ACM, 2017.

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

[115] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*, 2023.

[116] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. In *SIGCOMM*. ACM, 2015.

Appendix

A Convergence Analysis of MLT

This section formally presents a convergence analysis for MLT, essentially a distributed machine learning process with priority/selective dropping on gradients. Note that prior work [108] has already proven the comparable convergence rate of distributed learning with random dropping (i.e., unreliable network with independent and equivalent packet drop probability p for each message). Here, on top of [108], we further extend the convergence proof from random dropping to priority/selective dropping. The notations used here are shown in Table 3.

$\|\cdot\|$	l_2 norm for vectors
$\|\cdot\|_F$	the Frobenius norm of matrices
n	number of workers
m	number of servers
γ	model learning rate
p	packet dropping ratio

Table 3: Definitions and notations

The distributed optimization problem is defined as:

$$\min_{\vec{x}} f(\vec{x}) = \frac{1}{n}\sum_{i=1}^{n} f_i(\vec{x}), \tag{1}$$

where n is the number of workers, $f_i(\vec{x}) = \mathbb{E}_{\xi\ D_i} F_i(\vec{x}, \xi)$ represents the expected loss function F over D_i, the local data distribution of worker i. At each iteration, every worker performs SGD on a random chosen subset of dataset $D_t^{(i)}$:

$$G_t^{(i)} = \nabla F_i\left(X_t^{(i)}, D_t^{(i)}\right).$$

$X_t^{(i)}$, $G_t^{(i)}$ and $D_t^{(i)}$ denotes the model weights, generated gradients and training data of worker i at iteration t respectively.

Before sending the gradients, every worker i divides the gradients into m equal blocks:

$$G_t^{(i)} = \left((G_t^{(i,1)})^\intercal, (G_t^{(i,2)})^\intercal, \ldots, (G_t^{(i,m)})^\intercal\right).$$

When sending gradients $G_t^{(i)}$, some blocks may be dropped because of the networking condition and priority dropping. For each blocks, the gradients on every workers are collected and averaged by parameter server:

$$\widetilde{G}_t^j = \frac{1}{|N_t^{(j)}|}\sum_{i \in N_t^{(j)}} G_t^{(i,j)},$$

where $\widetilde{G}_t^j$ denotes the averaged gradients of block j at iteration t, and $N_t^{(j)}$ denotes the number of workers whose blocks j are successfully averaged at iteration t.

After averaging gradients, the parameter server updates the corresponding weight block using SGD algorithm and returns them back to each workers for their local updates. For workers that fail to receive the averaged block, they just use the original gradients. Formally, the updated gradients on worker i is:

$$X_{t+1}^{(i)} = \left((X_{t+1}^{(i,1)})^\intercal, (X_{t+1}^{(i,2)})^\intercal, \ldots, (X_{t+1}^{(i,m)})^\intercal\right),$$

where

$$X_{t+1}^{(i,j)} = \begin{cases} X_t^{(i,j)} - \gamma \widetilde{G}_t^j, & i \in \widetilde{N}_t^{(j)} \\ X_t^{(i,j)} - \gamma G_t^{(i,j)}, & i \notin \widetilde{N}_t^{(j)}. \end{cases}$$

$\widetilde{N}_t^{(j)}$ denotes the set of workers to which the averaged block j is successfully sent at iteration t.

For the algorithm, we make the following assumptions commonly used for analyzing stochastic optimization algorithms [64, 108].

Assumption 1. *We make the following commonly used assumptions:*

1. ***Lipschitzian gradient**: The gradient function $\nabla f_i(\cdot)$ is L-Lipschitian, which means*

$$\|\nabla f_i(\vec{x}) - \nabla f_i(\vec{y})\| \le L\|\vec{x} - \vec{y}\|$$

2. ***Bounded gradient**: The variance of stochastic gradient is bounded for every worker i and any $\vec{x}$.*

$$\mathbb{E}_{\xi \ D_i}\|\nabla F_i(\vec{x};\xi) - \nabla f_i(\vec{x})\|^2 \le \sigma^2, \forall i, \forall \vec{x}$$
$$\frac{1}{n}\sum_{i=1}^{n}\|\nabla f_i(\vec{x}) - \nabla f(\vec{x})\|^2 \le \xi^2, \forall i, \forall \vec{x},$$

3. ***Start from 0**: For simplicity, we assume $X_1 = 0$ w.l.o.g.*

With arbitrary packet dropping policy, the updated gradients on each worker can always be represented as the linear combination of local gradients.

$$X_{t+1}^{(i,j)} - X_t^{(i,j)} = G_t^{(\cdot,j)} W_t^{(j)},$$

where

$$G_t^{(\cdot,j)} := \left((G_t^{(1,j)})^\intercal, (G_t^{(2,j)})^\intercal, \dots, (G_t^{(i,j)})^\intercal\right).$$

$W_t^{(j)}$ is the coefficient matrix. And $\left[W_t^{(j)}\right]_{m,k}$ denotes the coefficient of worker m's gradients received by worker k after one update step. $\left[W_t^{(j)}\right]_{m,k} = 0$ means worker m's gradient block j is not received by worker k, which may be dropped either before or after the averaging during the communication with the parameter server.

[108] shows $W_t^{(j)}$ satisfies the following properties under uniformly random dropping environment:

$$\mathbb{E}[W] = \alpha_1 I_n + (1-\alpha_1)A_n \tag{2}$$
$$\mathbb{E}[W_t^{(j)} W_t^{(j)\intercal}]\alpha_1 I_n + (1-\alpha_1)A_n \tag{3}$$
$$\mathbb{E}[W_t^{(j)} A_n W_t^{(j)\intercal}] = \alpha_2 I_n + (1-\alpha_2)A_n \tag{4}$$

for some constants α_1 and α_2 satisfying $0 < \alpha_2 < \alpha_1 < 1$. While [108] considers the algorithm where workers perform the averaging operation, the properties also hold for dedicated parameter server setting. Also, as MLT adopts priority dropping mechanism, $(\alpha_1^{(j,t)}, \alpha_2^{(j,t)})$ varies in different blocks j and iterations t. To adopt the convergence proof in [108] for MLT, we use $\alpha_{1_{max}}, \alpha_{2_{max}}$ instead, which denotes the maximum value of $\max_{j,t}\alpha_1^{(j,t)}$ and $\max_{j,t}\alpha_2^{(j,t)}$ across all workers and iterations and preserve the validity of the proof. Thus we can get the following theorem:

Theorem 1. *(Convergence of MLT). Under Assumption 1, choosing learning rate γ to be small enough satisfying $1 - \frac{6L^2\gamma^2}{(1-\sqrt{\beta_{max}})^2} > 0$, MLT have the following convergence rate:*

$$\begin{aligned}
&\frac{1}{T}\sum_{t=1}^{T}\left(\mathbb{E}\|\nabla f(\vec{\overline{x}}_t)\|^2 + (1-L\gamma)\mathbb{E}\|\overline{\nabla} f(X_t)\|^2\right) \\
&\le \frac{2f(\vec{0}) - 2f(\vec{x}^*)}{\gamma T} + \frac{\gamma L\sigma^2}{n} + 4\alpha_{2_{max}}L\gamma(\sigma^2 + 3\xi^2) \\
&+ \frac{2\alpha_{2_{max}}L\gamma + L^2\gamma^2 + 12\alpha_{2_{max}}L^3\gamma^3)\sigma^2 C_1}{(1-\sqrt{\beta_{max}})^2} \\
&+ \frac{3(2\alpha_{2_{max}}L\gamma + L^2\gamma^2 + 12\alpha_{2_{max}}L^3\gamma^3)\xi^2 C_1}{(1-\sqrt{\beta_{max}})^2},
\end{aligned} \tag{5}$$

where

$$\nabla f(\vec{\overline{x}}_t) = \nabla f\left(\frac{1}{n}\sum_{i=1}^{n}\vec{x}_t^{(i)}\right)$$
$$\overline{\nabla} f(X_t) = \sum_{i=1}^{n}\nabla f_i(\vec{x}_t^{(i)})$$
$$\beta_{max} = \max_{j,t}(\alpha_1^{(j,t)} - \alpha_2^{(j,t)})$$
$$C_1 = \left(1 - \frac{6L^2\gamma^2}{(1-\sqrt{\beta_{max}})^2}\right)^{-1}.$$

It can be inferred from the definitions that $\beta = 1$ if and only if the dropping probability of the gradient block is 1, which may cause the bound to be infinity. In MLT we can make the assumption that no gradient block has dropping probability equal to 1, since the magnitude of gradients varies among different iterations.

By choosing appropriate learning rate $\gamma = \frac{(1-\sqrt{\beta_{max}})^2}{6L + 3(\sigma+\xi)\sqrt{\alpha_{2_{max}}T} + \frac{\sigma\sqrt{T}}{\sqrt{n}}}$, we can get

$$\begin{aligned}
\frac{1}{T}\sum_{t=1}^{T}\mathbb{E}\|\nabla f(\vec{\overline{x}}_t)\|^2 &\le \frac{(2f(\vec{0}) - 2f(\vec{x}^*) + L)\sigma}{\sqrt{nT}(1-\sqrt{\beta_{max}})} \\
&+ \frac{(2f(\vec{0}) - 2f(\vec{x}^*) + L)(\sigma+\xi)}{1-\sqrt{\beta_{max}}}\sqrt{\frac{\alpha_{2_{max}}}{T}} \\
&+ \frac{L^2(\sigma^2+\xi^2)}{(\frac{T}{n} + \alpha_{2_{max}}T)\sigma^2 + \alpha_{2_{max}}T\xi^2} \\
&+ \frac{(2f(\vec{0}) - 2f(\vec{x}^*))L}{T}
\end{aligned} \tag{6}$$

We can see from Equation 6 that the dominant term in the convergence rate ($O(1/\sqrt{nT})$) is consistent with prior works for both centralized SGD and decentralized SGD [65, 108], which theoretically prove that MLT will converge with the same order of iterations as the previous vanilla SGD methods.

B Thresholds Setting of Selective Dropping

This section presents the formulation to derive the optimal thresholds for the selective dropping mechanism (RED/ECN setting) in §4.2.2 by leveraging the queueing theory [16]. As a guidance from the analysis, to find the thresholds, we need to know the size of each layer of the model, and measure the additional rounds to model convergence caused by the loss of small/large gradients in each layer. Note that our analysis does not yet provide optimal thresholds for specific models, we leave the additional rounds measurement and optimal thresholds calculation as the future work. The notations used here are shown in Table 4.

N	number of switch queues
B	size of switch buffer
S_i	ECN/RED threshold of queue i
L_i	length of queue i
λ	packet arrival rate
μ	packer service rate
M	number of model layers
S_i^m	size of layer i in the model
θ	ratio of small gradients in the model
$f_i^S(.)/f_i^L(.)$	additional convergence rounds cost by 1% small/large gradient loss in layer i

Table 4: Definitions and notations

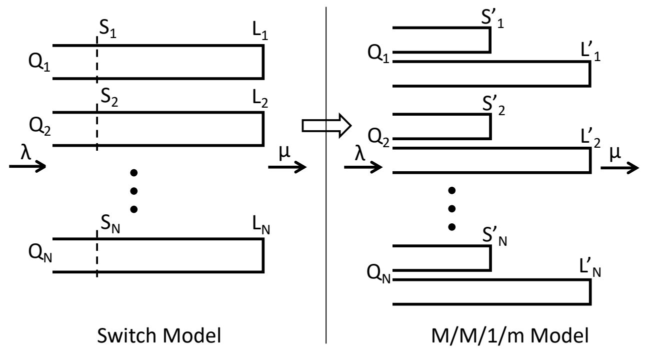


Figure 20: Problem formulation

Problem formulation: Figure 20 shows the mathematical modeling of MLT switch. All flows come with the arrival rate of λ, and enter the corresponding queues. For each queue, if the sum of queueing packets' size is larger than S_i, all small gradients packets come to this queue will be discarded. If the sum is larger than L_I, all packets will be discard. To simplify the analysis and take the advantage of M/M/1/m Model in queueing theory, we split each queue into two, the first is for small gradients only and the second is for large gradients only. The dropping thresholds for each one are S_i' and L_i'. We can easily represent S_i and L_i with S_i' and L_i': $S_i = S_i'/\theta, L_i = S_i' + L_i'$.

Then, we deduce the value of S_i' and L_i'. The arrival rate for one queue depends on the corresponding layers' packet arrival rate, for simplicity, we assume the rate is proportional to the size of the layer. Therefore, the arrival rate for queue i is $\frac{\lambda S_i^m}{S}$, and for the small gradients' queue, the value is $\theta\frac{\lambda S_i^m}{S}$, for the large one, is $(1-\theta)\frac{\lambda S_i^m}{S}$.

The service rate for one queue is determined on its priority, it is serviced only when the higher priority queues are idle, for the highest priority queue Q_1, the service rate is $\mu_1 = \mu$, the idle time is $1-\rho_1$, where $\rho_1 = \lambda/\mu$, for queue Q_2, the service rate is $\mu_2 = (1-\rho_1)\mu$. Generally, the idle time for queue Q_i is $1-\rho_i$, where $\rho_i = \lambda_i/\mu_i$ and the service rate is $\mu_i = \Pi_{k=0}^{i-1}(1-\rho_k)\mu$.

Supposed the service rate for small/large gradients is proportional to the size, thus, the service rate for small/large gradients queue in queue Q_i are $\mu_i^S = \theta\mu_i$ and $\mu_i^L = (1-\theta)\mu_i$ respectively. Therefore the idle time are $\rho_i^S = \lambda_i^S/\mu_i^S = \lambda_i/\mu_i = \rho_i = \rho_i^L$.

Suppose the losses of small and large gradients in each layer affect the convergence rounds independently, our goal is to minimize the impact of gradients' loss to model convergence, that is to find the optimal S_i and L_i to minimize the loss function $\sum_{i=1}^{N}\left(f_i^S(r_i^S) + f_i^L(r_i^L)\right)$, where r_i^S and r_i^L are the loss ratio for small/large gradients in queue Q_i. There are a lot of existing solutions to solve the optimization problem, e.g. gradient descent. Here, we only need to calculate the value of r_i^S and r_i^L. In fact, for each queue, we can treat it as a typical M/M/1/m model in queueing theory, specially, one FIFO queue with finite capacity. Previous work [16] has derived the calculation formula of loss rate, that is $\frac{\rho^m - \rho^{m+1}}{1-\rho^{m+1}}$, where ρ is the idle time of the queue. Therefore, we have: $r_i^S = [(\rho_i)^{\theta S_i} - (\theta\rho_i)^{S_i+1}]/[1-(\rho_i)^{\theta S_i+1}]$, $r_i^L = [(\rho_i)^{L_i-\theta S_i} - (\rho_i)^{L_i-\theta S_i+1}]/[1-(\rho_i)^{L_i-\theta S_i+1}]$, then we express the loss function in terms of known parameters and thresholds (S_i, L_i), after solving this optimization problem, we obtain the optimal thresholds.

C Supplemental Experiments

Due to the space limitation, in the main part of the paper, we only show the results of VGG16 and Transformer in Figure 11, 12, 16, 17 and 18. Here, we append the results of ResNet50 and GoogleNet in Figure 21, 22, 23, 24, 25 and 26. Please note that the results/trends embodied in these figures are consistent with that in the main part of the paper.

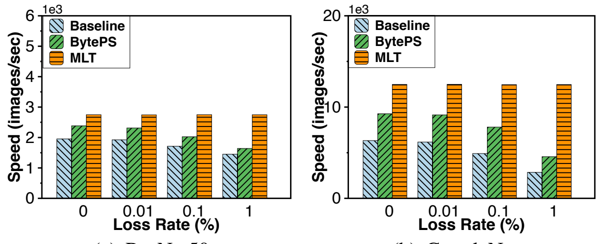


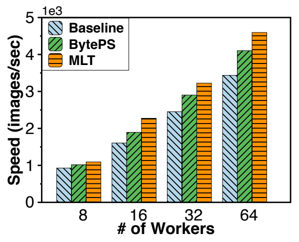
Figure 21: Speedup under gray failure of links.

D Open Questions and Discussion (Cont'd)

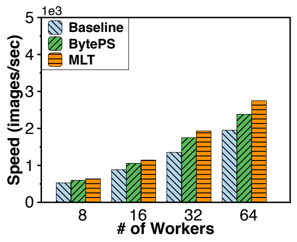
Co-existing with non-DNN traffic. This paper focuses on AI-centric networking (AICN) dedicated to DNN training. However, in datacenter with multi-purpose traffic, our implementation of MLT should mitigate its intrusion on non-DNN

Model	GoogleNet [93] (13%)	LSTM [42] (11%)	ResNet34 [41] (14%)	Wide ResNet50 [109] (17%)
	AlexNet [57](18%)	VGG16 [91] (17%)	GRU [26] (12%)	ResNet50 [41] (16%)
	EfficientNetB0 [94] (17%)	VGG19 [91] (18%)	Wide ResNet101 [109] (16%)	ShuffleNetV2 [71] (16%)
	VGG13 [91] (16%)	ResNet18 [41] (18%)	DenseNet169 [47] (16%)	ResNet101 [41] (17%)

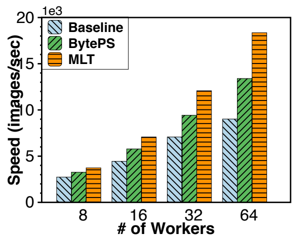
Table 5: The loss-tolerant bounds measurement with MLT, under the condition that the models converge with the same iterations to the same accuracy. We use the same models and settings with Table 1.



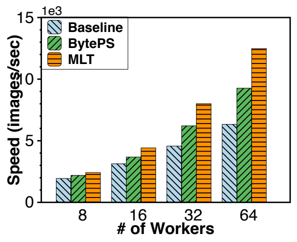
(a) MXNet, ResNet50



(b) PyTorch, ResNet50

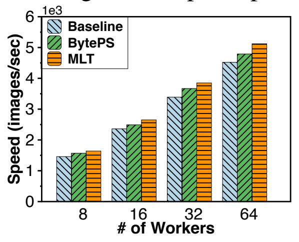


(c) MXNet, GoogleNet

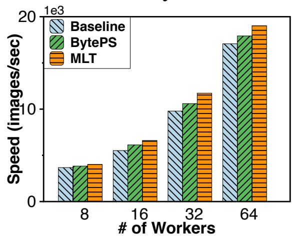


(d) PyTorch, GoogleNet

Figure 22: Speedup with MXNet and PyTorch.

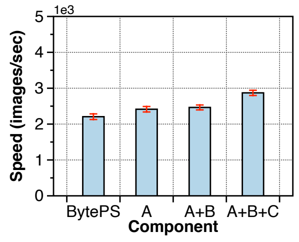


(a) Ring All-reduce, ResNet50

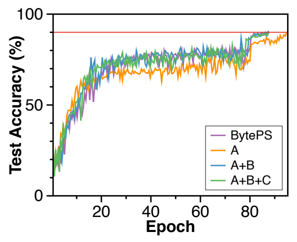


(b) Ring All-reduce, GoogleNet

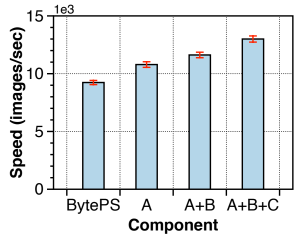
Figure 23: Speedup with Ring All-reduce



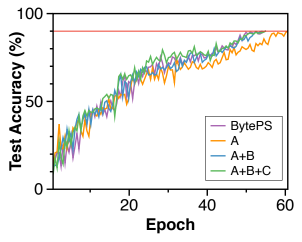
(a) ResNet50, Speed



(b) ResNet50, ETA

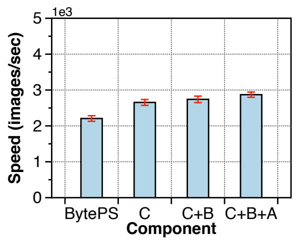


(c) GoogleNet, Speed

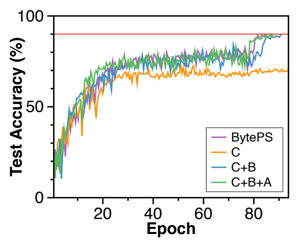


(d) GoogleNet, ETA

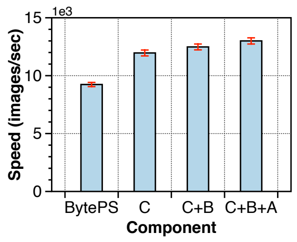
Figure 24: Effectiveness of design components (I).



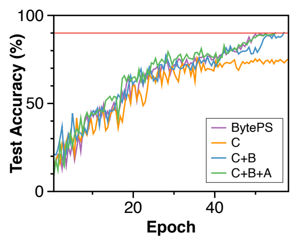
(a) ResNet50, Speed



(b) ResNet50, ETA

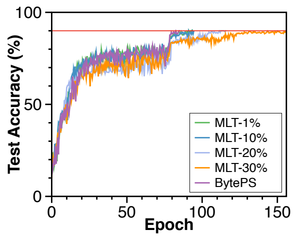


(c) GoogleNet, Speed

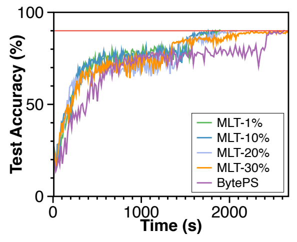


(d) GoogleNet, ETA

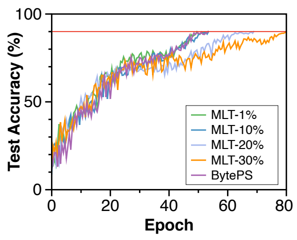
Figure 25: Effectiveness of design components (II).



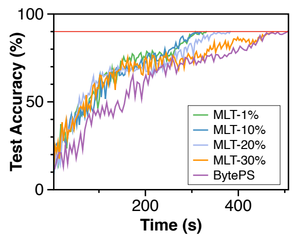
(a) ResNet50, ETA



(b) ResNet50, TTA



(c) GoogleNet, ETA



(d) GoogleNet, TTA

Figure 26: Impact of loss-tolerant bounds.

traffic and ensure bandwidth fair-sharing. A straightforward idea is to separate DNN and non-DNN traffic into different queues and perform fair-queueing between them. One issue to consider is that current commodity switches only have a limited # of queues (typically 8), so the # of queues dedicated for MLT become even smaller. For priority queueing, it is not a concern because the recent design [68] of using two priority queues to emulate many fine-grained priorities can be borrowed: the insight is that we only need to privilege the highest priority flows; MLT can leverage such idea and only place the foremost layers into the highest priority queue each time. On the other hand, it may reduce the granularity of selective dropping of gradients across DNN layers, as this relates to the # of physical queues available to MLT.

Flexible selective dropping. We consider the gradients of back layers more *important* under the assumption of pre-training (§3.3). As researchers in ML area are improving the model architectures and training algorithms, the loss tolerance across model parameters may vary in the future. MLT allows a flexible selective dropping policy through tagging. One can simply modify the gradient tagging policy at end-hosts to fit new features in the training process.

MLT vs SRD. AWS SRD [89] is a hardware transport protocol that also uses packet spraying to avoid hotpots. It provides reliable but out-of-order delivery. To be used by general-purpose applications, SRD still relies on a messaging layer above it to restore orders. In contrast, MLT exploits ML-specific properties to provide semi-reliable delivery without the need of packet order restoration.