OptiReduce: Resilient and Tail-Optimal AllReduce for Distributed Deep Learning in the Cloud

Ertza Warraich, Omer Shabtai[†], Khalid Manaa[†], Shay Vargaftik[‡],
Yonatan Piasetzky[†], Matty Kadosh[†], Lalith Suresh[§], Muhammad Shahbaz[*]
Purdue University [†]*Nvidia* [‡]*VMware Research* [§]*Feldera* [*]*University of Michigan*

Abstract

We present OptiReduce, a new collective-communication system for the cloud with bounded, predictable completion times for deep-learning jobs in the presence of varying computation (stragglers) and communication (congestion and gradient drops) variabilities. OptiReduce exploits the inherent resiliency and the stochastic nature of distributed deep-learning (DDL) training and fine-tuning to work with approximated (or lost) gradients—providing an efficient balance between tail performance and the resulting accuracy of the trained models.

Exploiting this domain-specific characteristic of DDL, OptiReduce introduces (1) mechanisms (e.g., unreliable bounded transport with adaptive timeout) to improve the DDL jobs' tail execution time, and (2) strategies (e.g., Transpose AllReduce and Hadamard Transform) to mitigate the impact of dropped gradient entries on model accuracy. Our evaluation shows that OptiReduce achieves 70% and 30% faster time-to-accuracy (TTA), on average, when operating in shared, cloud environments (e.g., CloudLab) compared to Gloo and NCCL, respectively.

1 Introduction

Synchronous distributed data-parallel training [175] is now the de-facto standard for training and fine-tuning large-scale deep-learning models (comprising billions or even trillions of parameters) and datasets (comprising terabytes of data) that form the backbone of many mainstream enterprise applications, including computer vision [64, 74, 96, 161], natural-language processing and large-language models [53, 63, 110, 167], recommendation and prediction systems [70, 71, 84, 86, 130], and healthcare [93, 112, 125, 169]. Under this scheme, the training occurs in rounds (or epochs). Workers locally train a copy of the model on a fragment of data and then share the model updates (i.e., gradients) among themselves over the network to compute an aggregated result. The aggregate is then used to update the model locally for the next round of training. Distributed deep-learning (DDL) is, therefore, inherently a computation- and communication-intensive workload and is becoming even more so with the growing model sizes (e.g., Bart [101], GPT-2/3 [53, 126], LLaMA [65, 146]), and datasets [38, 135, 143].

To train and fine-tune such large models, extensive efforts are underway in reducing both the computation and communication time of DDL jobs, albeit in isolation. On the one hand, we have GPUs [141] and emerging hardware accelerators, like Tensor Processing Units (TPUs) [90], that are drastically bringing down the computation time—reducing it by 62× over the last decade [136]. While, on the other hand, we have recent proposals based on programmable switches [157] that aim at reducing the communication time by 2–5× (via in-network aggregation) [136]. Yet, when seen together, both these efforts mainly help in improving the average completion time of a deep-learning job (either by accelerating computation or communication). The vast array of system-level variabilities (e.g., device failures, OS and hypervisor scheduling, and resource contention) and network-level delays (e.g., congestion and retransmissions) still lead to long tails; hence, resulting in poor overall performance for these training jobs—with tail reaching as high as 4× the mean latency in shared environments (e.g., public cloud providers)[1] [72, 77, 99, 106, 118, 153, 166].

In this paper, we make the case for OptiReduce, a collective-communication system for the cloud tenants that ensures bounded, predictable completion times for deep-learning jobs in the presence of myriad computation and communication variabilities. Public clouds are becoming increasingly appealing for training, and more specifically fine-tuning, large foundation models [50], for enterprises and individuals lacking resources to set their own in-house distributed training clusters [14, 16, 17, 20, 21, 28, 33, 36, 37]. OptiReduce exploits the inherent resiliency and the stochastic nature of deep-learning systems to work with approximated or lost gradients and provides an efficient balance between tail performance and the resulting accuracy of the trained models. Others are already utilizing this characteristic of deep learning to optimize DDL hardware design (e.g., chip area [134, 171]), minimize traffic overhead [68, 103, 109, 154], or offload certain DDL tasks to the network switches [99, 136, 153, 157, 160]. For instance, to reduce communication time, ATP [99] and SwitchML [136] leverage fixed-point arithmetic for gradient aggregation in programmable switches with acceptable approximation loss, whereas MLT [153] prioritizes and drops packets inside switches based on model layers and gradient magnitudes to limit loss in accuracy. Various gradient-compression schemes [68, 103, 109, 154] employ lossy compression to reduce network traffic over-

[1] Cloud providers typically do not offer preferential treatment to small tenants, but even large tenants with dedicated racks face long tails when communicating across racks in the provider's network. Private communication with a hyperscaler.

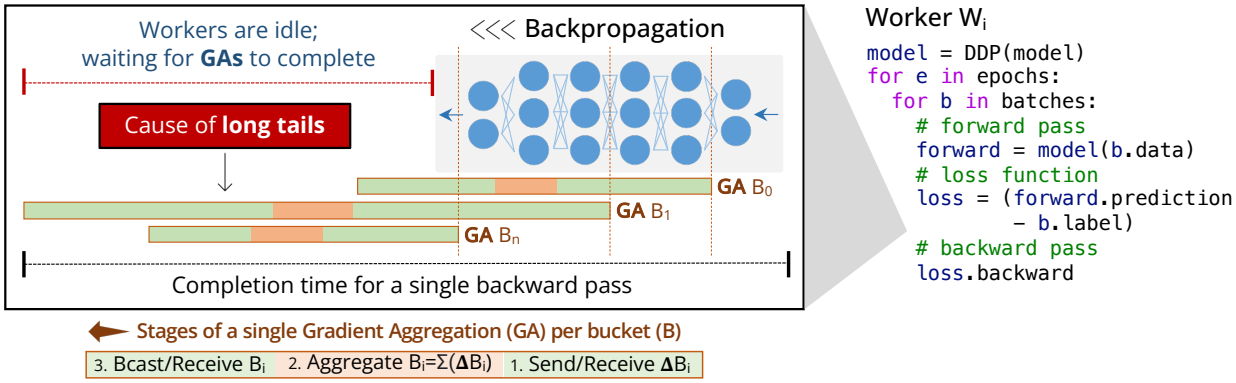


Figure 1: A backpropagation pass in distributed data-parallel (DDP) training. Multiple gradient aggregation (GA) runs share a bucket (B_i) worth of gradient entries among worker nodes (W_n), in parallel. The two send(bcast)/receive stages (1, 3) in GA incur the most time—contributing to the tail latency and stalling workers.

head (e.g., total bytes transferred) while limiting deviation from the achievable model accuracy. Similarly, hardware designers are incorporating approximate operations (e.g., approx. multipliers [134, 171]) in their architectures to minimize resource and energy usage—to scale to ever-increasing DDL models. However, these solutions are still susceptible to tail effects (e.g., slow workers and network variabilities) [47, 61, 78, 108, 127, 140, 153, 170, 173], and are not optimized for cloud environments, often times requiring direct access to the provider's network infrastructure.

In OPTIREDUCE, we exploit this resiliency and replace the (tail-prone) deterministic, run-to-completion stages of an AllReduce collective in DDL, with best-effort, *time-bounded* implementations.

- OPTIREDUCE introduces a *Transpose-Allreduce Collective (TAR)* to reduce the impact of lost gradient entries by establishing direct peer-to-peer communication among nodes in each round, rather than propagating the entries through all nodes as in Ring [121].
- It also implements an *Unreliable Bounded Transport (UBT)* to maximize the number of gradient entries received during each window. UBT introduces the notion of *adaptive timeout* to restrict the time a deep-learning job spends doing computation (aggregation) and communication (gradient sharing). Furthermore, it adds support for *dynamic incast* to dynamically adjust the number of concurrent senders per receiver in each round, thus optimizing communication by reducing the total rounds required for gradient aggregation.
- Lastly, to minimize the impact of missed or dropped gradient entries, OPTIREDUCE employs the *Hadamard Transform (HT)* [124] to ensure, for any drop pattern (e.g., tail drops), a receiver still obtains an unbiased estimate of the model's gradients resulting in a minimal loss in accuracy.

We implement OPTIREDUCE as a new AllReduce scheme inside Gloo [13],[2] a popular collective-communication library. Doing so makes OPTIREDUCE immediately compatible with existing DDL frameworks (e.g., PyTorch) without modifications. We run our experiments on various popular large deep-learning models (including BART [101], OpenAI's GPT-2 [126], and Meta's Llama 3.2) and evaluate OPTIREDUCE on CloudLab [66]—a public cloud facility for researchers—as well as under different shared environment settings using a local virtualized cluster, with varying tail-to-median latency ratios. We have made our complete OPTIREDUCE prototype [29] publicly available at https://optireduce.github.io.

Our evaluation demonstrates that OPTIREDUCE achieves, on average, 57% and 25% faster time-to-accuracy (TTA) on CloudLab compared to Gloo [13] and NCCL [87], respectively. These performance gains are even more pronounced in environments with larger tail-to-median latency differences, where OPTIREDUCE outperforms Gloo by 91% and NCCL by 35% (§5). We observe that it is the latency of the gradient aggregation (GA) step (Figure 1) that inflates three folds when operating under tail-heavy environments (e.g., public clouds), hence strengthening the need for a new collective like OPTIREDUCE. We also perform a deeper analysis of the various components of OPTIREDUCE. For example, in our evaluation, enabling Hadamard Transform mitigates the impact of tail-drops and improves TTA by 1.8× even when up to 10% of gradient entries are lost.

We begin with a background on distributed deep-learning (DDL) training—more specifically, distributed data-parallel (DDP) training—and the impact of stragglers on performance (§2). We then make a case for and present a design (§3) and implementation (§4) of a new communication-collective system, OPTIREDUCE, and evaluate how it exploits DDL's resiliency against gradient loss to improve performance (§5).

2 Background & Motivation

2.1 Distributed Deep Learning & Stragglers

Distributed deep learning (DDL) helps scale (and speedup) model training by utilizing an increasing number of hardware

[2]We pick Gloo for its ability to use both GPUs and CPUs (more suitable for a cloud environment), but OPTIREDUCE can operate with other libraries as well (e.g., NCCL [87] or MSCCL [24]).

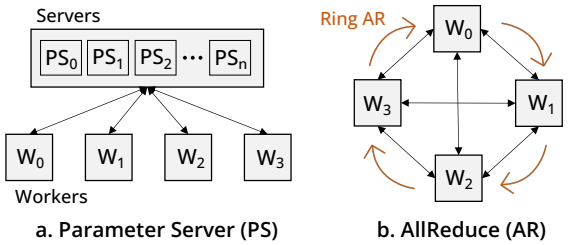


Figure 2: Architectures for gradient aggregation: Parameter Server (PS) and AllReduce (AR).

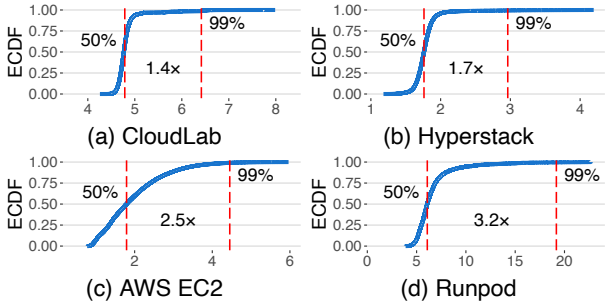


Figure 3: The latency ECDF (in milliseconds) showing tail-to-median ratio ($P_{99/50}$) observed across leading AI cloud platforms.

accelerators, e.g., GPUs [141] and TPUs [90], across server nodes [39, 55, 116]. To do so, it employs two approaches: *(i)* distributed data parallelism (DDP) [175] to run batches[3] on multiple accelerators in parallel, and *(ii)* distributed model parallelism (DMP) [91] to deploy larger models that fail to fit on a single accelerator. These approaches are orthogonal and can be used in conjunction. We focus on DDP in this paper to highlight our contributions (§3).

In distributed data parallelism (DDP), multiple worker nodes run the same deep-learning model on their portion of a training dataset, distributed evenly across nodes. Each portion is further subdivided into batches, which are processed sequentially (i.e., forward pass, loss function, and backward pass) in each epoch, Figure 1. During the forward pass, the model (e.g., a neural network) operates on a batch and generates a prediction, which is then compared with the ground truth (e.g., label) to calculate the model's loss. Next, the backward pass computes gradients using a loss function, which is used by an optimization algorithm (e.g., Stochastic Gradient Descent, SGD [52, 94]) to update the model parameters. Finally, to ensure all workers learn from what others have learned from their portion of the dataset, these gradients are averaged (reduced) and shared across all nodes, after each backward pass.

The process of accumulating, reducing, and sharing gradients back with worker nodes is referred to as *gradient aggregation* (or reduction) [62]. Originally, reduction used to happen strictly after the backward pass; however, more recently, to hide communication latency, modern frameworks (like PyTorch [106]) overlap it with the backward pass (Figure 1). As soon as a bucket (B) worth of gradient entries becomes available on a node, it is sent for reduction.[4]

Two common architectures for gradient aggregation are: Parameter Server (PS) [105] and AllReduce (AR) [59]. In the PS architecture (Figure 2a), a central server (or group of servers) receives (gathers) gradients from participating workers, aggregates (reduces) them, and broadcasts them back to all nodes. In AR (Figure 2b), instead of having separate servers, we distribute the aggregation task across workers, each reducing a subset of gradients and distributing them among themselves (e.g., Ring [121]). Both these architectures have their pros and cons. PS operates well in environments with less powerful worker machines but is bandwidth hungry—increasing linearly with the number of worker nodes. AR, especially Ring, is bandwidth-optimal but leads to longer execution delays that inflate with the number of worker nodes.

Impact of Stragglers on Performance. As shown in Figure 1, during each backward pass, all DDP worker nodes wait for the gradient aggregation (GA) operations to complete before processing the next batch of data. The forward and backward passes computation mostly takes place on a machine-learning (ML) accelerator (e.g., GPU or TPU)—a highly parallel and pipelined architecture with predictable and bounded execution time [69, 156]. Therefore, it is typically the GA operations that lead to long tails and GPU stalls (taking as much as 50% of the overall DDP processing time) [136, 137]. Our measurements across major AI cloud platforms—including AWS EC2 [3], Hyperstack [17], CloudLab [8], and RunPod AI [34]—quantify network tail latencies in a distributed training environment. Using the Gloo benchmark [12] with 2K gradients on eight nodes, we observe tail-to-median ($P_{99/50}$) latency ratios reaching up to 3.2× (Figure 3).

Various factors can contribute to this slowdown in gradient aggregation, including slow workers, transmission delays, incast effects, packet loss and retransmissions, network congestion, and more. For example, even in the PS architecture, each node sends a complete set of gradients to the parameter server, which can result in excessive drops and retransmissions, due to high incast at the ToR switch [104, 172]—hence, increasing the time to process gradients. Similarly, in Ring, a single slow worker (or a buggy link) can cause significant delays, because all nodes participate in the aggregation operation in the form of a ring.

2.2 Straggler Mitigation & Gradient Loss

Mitigating stragglers in distributed systems (as well as distributed deep learning) is an active area of research [48, 54, 60, 61, 81, 92, 132, 145, 158, 164]. One direction focuses on treating stragglers as black boxes and employs schemes, such as redundant task execution [73, 159] or skipping slow workers [57, 67, 164], to mitigate the delays due to network congestion or heterogeneous hardware, for example. They either employ backup workers and select the output of the fastest ones,

[3] A batch is a set of training data used in a given forward/backward pass.
[4] PyTorch limits these simultaneous reduction operations to two [106].

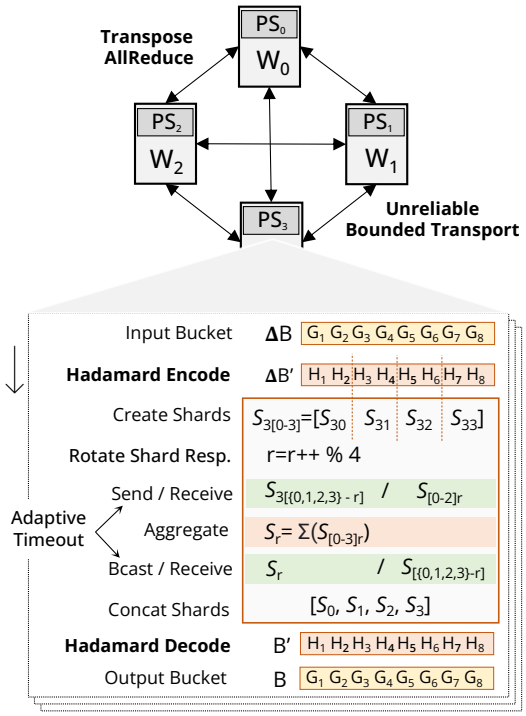


Figure 4: The OPTIREDUCE design: Transpose AllReduce with colocated parameter servers, Unreliable Bounded Transport, and Hadamard Transform.

or simply skip the slow workers altogether. However, the former can significantly increase the operational expense (in dollars). For example, training a GPT-3 model consisting of 175 billion parameters over 355 GPU-years (on a V100) [5, 53] can cost an additional $1 Million ($5.6 Million total) on the AWS instance, `p3.16xlarge`, [27] with only 16% backup nodes (i.e., 2 backups for every 10 worker nodes). This cost can further inflate by about 10× when using more powerful GPUs (e.g., A100 and H100) [6, 7], higher link speeds (e.g., 40/100 Gbps), and RDMA-enabled NICs [4]. Whereas ignoring worker nodes entirely, in the latter case, can lead to slower convergence rates and poor accuracies [57, 164]. The other direction is to replace commodity servers with specialized hardware (e.g., powerful machines with predictable performance [163, 174]) and dedicated (lossless) communication fabric [47, 95, 123]. Despite their success in HPC-like environments [117, 120, 122, 162], these solutions are not applicable in a cloud environment with myriads of tenants, all sharing the resources of the underlying data centers.

Instead of treating them as black boxes or specialized devices, we argue to replace the (tail-prone) deterministic, run-to-completion workers with their best-effort, *time-bounded* implementations. The idea is to restrict the processing time of a slow worker and utilize its partial output (gradients) in the next training phase, rather than skipping it entirely.

Resilience to Gradient Loss vs. Performance and Accuracy. Unlike traditional distributed systems (e.g., file sharing and web serving), the stochastic nature of distributed deep-learning systems provides an interesting trade-off between gradient loss (approximation or drops), performance, and accuracy. These systems (based on SGD-based optimization) [52] are shown to be resilient against estimation inaccuracies in stochastic gradients under different settings [79, 153]. For example, various gradient sparsification [68, 131, 154] and quantization [44, 109, 155] schemes employ this fact to reduce network traffic overhead. ATP [99] and SwitchML [136] utilize fixed-point arithmetic to execute gradient aggregation in programmable switches with acceptable approximation loss. Hardware designers incorporate approximate operations (e.g., approx. multipliers [134, 171]) to minimize chip area and energy usage. More recently, MLT [153] demonstrated that these deep-learning models (e.g., CNNs and LMs) are also resilient to a certain degree of gradient drops—sustaining high accuracy up to 1% of gradient loss. Additionally, the impact of gradient loss further diminishes as the number of worker nodes increases [168].

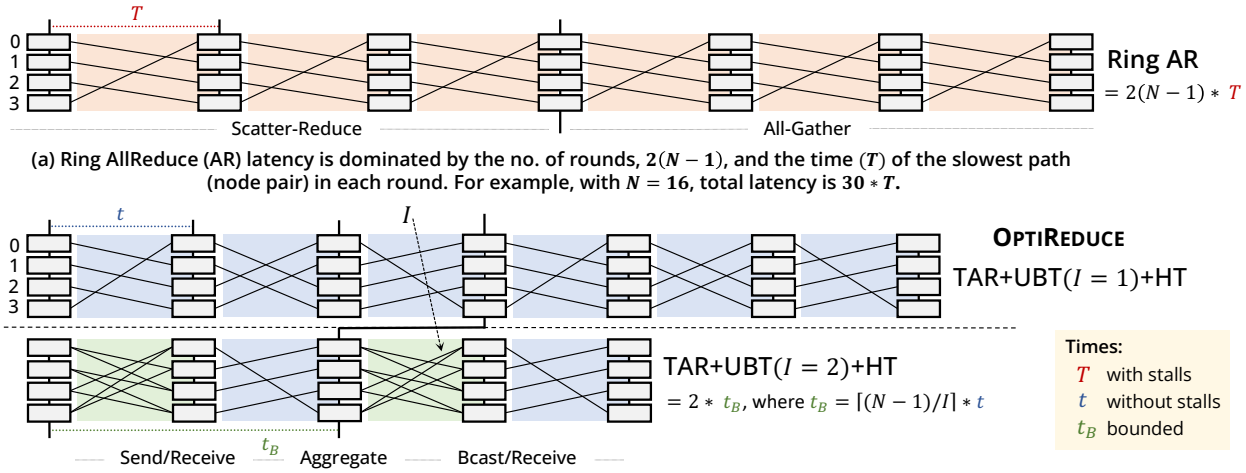
3 Design of OPTIREDUCE

We present OPTIREDUCE, a robust AllReduce communication collective system optimized to mitigate tail-latency by exploiting the unique characteristics of distributed-deep learning (DDL)—i.e., resiliency to gradient loss—to quickly reach the convergence accuracies of traditional architectures (e.g., PS and Ring) while mitigating the impact of stragglers and network variabilities.

Overview. Figure 4 shows the various components of OPTIREDUCE. *Transpose AllReduce (TAR)* (§3.1) implements a peer-to-peer collective-communication fabric, where each node also serves as a parameter server (PS)—a colocated PS architecture [68, 88]. *Unreliable Bounded Transport (UBT)* (§3.2) allows these nodes to connect with each other in a best-effort but controlled manner, and bounds the time spent by the two send(bcast)/receive stages during the AllReduce phase. The PS nodes encode (and decode) gradients in the input bucket (B), using *Hadamard Transform (HT)* (§3.3) to disperse the effect of gradient loss, before creating shards (S_{ij}) of gradients to be sent to other nodes for reduction. They iteratively *rotate shard responsibility* among themselves by maintaining a global index (r). Finally, OPTIREDUCE employs mechanisms (i.e., snapshots and selective skipping) to protect against excessive gradient loss due to transient errors or failures (§3.4); these safeguards ensure robustness and help maintain accuracy under unstable conditions. All these components operate in tandem to optimize for three competing objectives in OPTIREDUCE: maximizing performance, minimizing gradient drops, and sustaining accuracy.

3.1 Transpose AllReduce (TAR)

We begin with Transpose AllReduce (TAR), which implements a hierarchical peer-to-peer gradient-sharing strategy to limit the impact of lost gradient entries when applying the other design optimizations, discussed later. TAR operates by



(a) Ring AllReduce (AR) latency is dominated by the no. of rounds, $2(N-1)$, and the time (T) of the slowest path (node pair) in each round. For example, with $N = 16$, total latency is $30 * T$.

(b) OPTIREDUCE improves latency by reducing the no. of rounds to $2\left\lceil\frac{N-1}{I}\right\rceil$ with incast parameter (I) and bounds the path delay to t_B using Transpose AllReduce (TAR), Unreliable Bounded Transport (UBT), and Hadamard Transform (HT). For example, with $N = 16, I = 2$, the total latency is within $2 * t_B = 16 * t$, where $t \ll T$.

Figure 5: A comparison of Ring versus OPTIREDUCE.

having each node send its gradient entries directly to all other nodes during an AllReduce phase for aggregation; hence, a lost entry would only impact the aggregated results of a given node-pair in that phase. Whereas, in Ring [121], the impact is accumulated and propagated through a ring until it reaches the intended destination nodes. For instance, in our micro-benchmarks (§5.3), the Mean Squared Error (MSE) between the expected gradients and those of Ring in the presence of loss is 6× that of TAR.

3.1.1 TAR Algorithm: A Colocated PS-inspired Collective. TAR combines the key features of traditional AllReduce (i.e., P2P communication) [59, 102] and Ring (i.e., minimizing bandwidth using shards and avoiding incast via rounds) [121].

In TAR, each node acts as a worker as well as a parameter server (PS), connected together over a P2P collective-communication fabric (Figure 4). The i^{th} PS node (PS_i) receives a bucket of gradient entries (ΔB) as input from the worker process (W_i) and divides it into N shards ($S_{i[0\ldots N-1]}$), equal to the number of nodes. Keeping the r^{th} shard it is responsible for aggregating (S_{ir}), the node PS_i sends the remaining shards ($S_{i[\{0\ldots N-1\}-r]}$) to the neighboring nodes. At the same time, PS_i waits for its shards ($S_{[\{0\ldots N-1\}-i]r}$) and aggregates (i.e., averages) them with S_{ir} into a single shard S_r. Next, PS_i broadcasts S_r to all nodes and receives the aggregated shards from them, $S_{[\{0\ldots N-1\}-r]}$. Finally, these shards are concatenated into a bucket (B) and forwarded to the worker (W_i) to process the next batch of data. When $r = i$, the whole operation appears like a row-wise sum of the transpose of the shard matrix S, as shown in Figure 6; hence, the name *Transpose* AllReduce.

In P2P, each PS node communicates directly with the other nodes (instead of forming a ring). When sharing gradients, they all interact with each other twice: once sending/receiving shards to the other node and then broadcasting/receiving aggregated shards, hence limiting the impact of accumulated

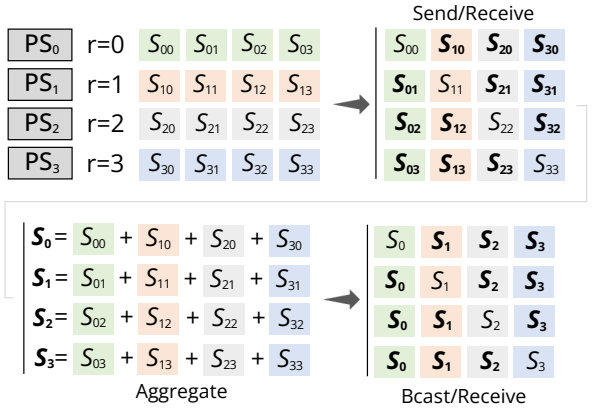


Figure 6: Transpose AllReduce algorithm: PS nodes send/receive shards S_{ij}, aggregate them, and bcast/receive to other nodes (all acquiring the same copy).

gradient loss. Sharding also alleviates the load on the PS nodes, where each node only aggregates a bucket (B) worth of gradients (rather than $N * B$).[5] Moreover, TAR utilizes the same bandwidth as Ring by sending $B * (N-1)$ bytes over the network during the two send(bcast)/receive stages. Lastly, to mitigate incast, TAR splits the communication between PS nodes over multiple rounds, where—unlike Ring with fixed node-pairs (Figure 5a)—nodes communicate with each other using a round-robin strategy, ensuring a given node-pair never repeats across rounds (Figure 5b).

3.1.2 Hierarchical 2D TAR: Scaling to Larger Node Clusters. TAR scales efficiently using a hierarchical design similar to the 2D Ring [22, 147]. By grouping nodes, it first performs intra-group communication, in parallel, to locally aggregate gradients, followed by inter-group communication for global aggregation. Doing so reduces the number of rounds from $2(N-1)$ in traditional TAR to $2(N/G-1)+(G-1)$, where

[5] PyTorch and TesnorFlow typically use a bucket size of 25 MB [106, 133].

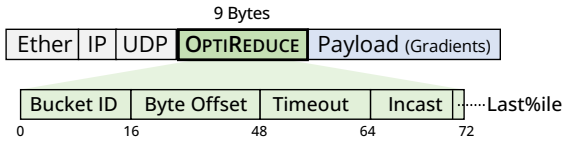


Figure 7: OptiReduce's header format

N is the total number of nodes and G is the number of groups. For example, with $N = 64$ and $G = 16$, traditional TAR requires an order of magnitude more rounds than 2D TAR—126 compared to just 21. We provide more details in Appendix A.

Summary: TAR functions similarly to Ring, yet it *reduces the impact of lost gradients* by establishing P2P communication among all nodes in each round, avoiding the propagation of losses via aggregation through intermediate nodes.

3.2 Unreliable Bounded Transport (UBT)

One of the primary causes affecting tail latency in DDL is the variability in the network behavior due to congestion [61, 153]. Current transport protocols (like TCP) further exacerbate these effects by demanding reliable, in-order delivery of packets (gradients) between training nodes—if packets are dropped or received out-of-order, TCP will stall until all gradients are received over the affected path.

However, simply replacing TCP with message-based protocols (like UDP) would not work. While UDP is faster, as it avoids packet retransmission and reordering, it lacks congestion control, which can lead to network congestion collapse. Moreover, UDP sends data at full link speed (e.g., 100 Gbps), causing excessive drops (loss of gradients) beyond what DDL models can tolerate.

To address this, we enhance UDP with adaptive timeouts, dynamic incast, and minimal rate control to create a new Unreliable Bounded Transport (UBT) protocol, which limits its computation and communication time while maximizing gradient delivery in each round. It adds a new 9-byte header, `OptiReduce` (Figure 7), to commit arriving packets (with gradients) to the right bucket and offset using the header fields, `Bucket ID` and `Byte Offset`, respectively. These fields ensure that gradients reach the correct bucket, irrespective of the ordering of the incoming packets when multiple GA operations are running in parallel (Figure 1).

3.2.1 Adaptive Timeout. UBT implements adaptive timeouts to bound the tail communication time of the send(bcast)/receive stages of the GA operations to t_B (Figure 5b). By restricting the time to t_B, we can control the worst-case execution of these stages—allowing GA operations to finish within a bounded time.

However, there are a couple of challenges with this approach. (1) How to select the value of t_B? Too small will lead to undue loss, and too high will cause unnecessary delays. Moreover, the value will vary with environmental settings (e.g., GPU type, CPU clock, vCPUs, and interface speed) and parameters (e.g., no. of nodes, bucket sizes, and incast). (2) A single lost packet, which is likely in UBT, would cause the

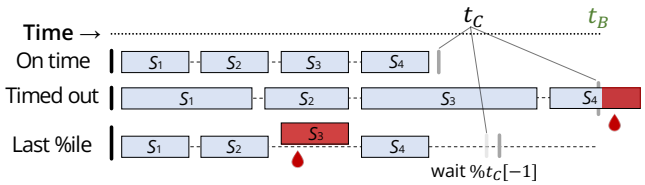


Figure 8: Different timeout strategies in OptiReduce.

GA operation to always take t_B (worst-case) time to finish. TCP, on the other hand, can perform better in some instances where communication may finish faster than waiting for the full timeout (t_B), even with retransmissions.

Selecting the Timeout Value (t_B). As shown in Figure 1, during backpropagation, multiple GA operations execute in parallel on buckets of varying sizes. For selecting t_B, during the initialization phase, we run GA with TAR and TCP, using the largest bucket, for a couple of iterations to collect completion times for both send(bcast)/receive stages. PS nodes share these values with each other using the `Timeout` field in the OptiReduce header (Figure 7).

We then form a list of these times and set t_B to the 95th %ile of that list. In §5, we show that using 20 iterations and the 95th %ile value allows OptiReduce to sustain full model accuracies while finishing up to 2× faster.

Progressing Quickly via Early Timeout. To avoid approaching t_B every time a loss happens, we introduce an early timeout scheme, which causes GA's receive stages to expire whenever there are no remaining gradient entries to read (i.e., the buffer is empty). For each bucket, we track a moving average (t_C) of completion times; we keep separate averages for both the receive stages in GA (Figure 5). The sender PS node tags the last 99th %ile packets by setting the `Last%ile` field in the header. When the buffer is empty, the receiver node checks if some of the last %ile packets have been received from all nodes. If so, it waits for an x% of t_C time before expiring (Figure 8).

The value of x% is dynamically adjusted based on the percentage of gradient entries dropped from the previous round. Starting at 10%, the goal is to maintain gradient losses between 0.01% and 0.1%. If losses exceed this range, x% is doubled until they return within the limit. If losses drop below 0.01%, x% is decreased by 1 until the desired range is reached. (The maximum x% is capped at 50%.) If gradient losses exceed 2% at any point, we activate Hadamard Transform (§3.3) to mitigate the effects of dropped gradients on convergence accuracy. [6]

We calculate t_C in the following steps. First, we compute the (expected) completion time of a given receive stage: (1) if on time, then we set t_C to the current time spent, (2) if timed out, then $t_C = t_B$, and (3) if last %ile received, then t_C is set to the expected time needed to receive all data (e.g., t_C = current time spent × total / received data). Next, we pick the median

[6] The 2% threshold is set based on prior work [153] and our evaluations §5.3.

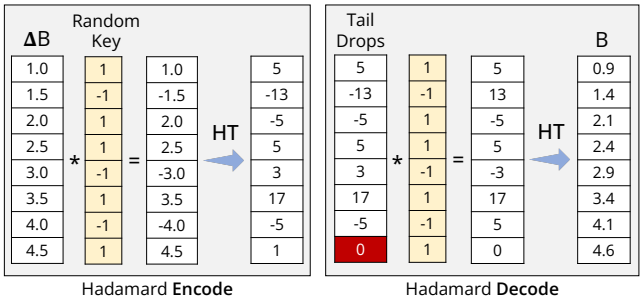


Figure 9: Dispersing the effect of lost gradients (e.g., due to tail drops) using Hadamard Transform (HT).

t_C from the values computed by the N PS nodes (shared over the `Timeout` field in the header). Finally, we calculate the moving average: $t_C = \alpha * t_C + (1-\alpha) * t_C[-1]$.

3.2.2 Dynamic Incast. UBT further introduces a notion of dynamic incast (Figure 5b). The TAR's P2P communication model lets OptiReduce alter the number of senders (I) a PS node can receive gradients from in a given round. For example, setting $I = 1$ (a single sender) would cause TAR to take the same number of rounds as Ring, $2(N-1)$; however, increasing $I = 2$ would quickly reduce these rounds by about half, $2\lceil (N-1)/2 \rceil$; and so on.

The incast parameter (I) can be configured either statically at boot time, based on the available network and node capacity (e.g., modern datacenters can handle hundreds of thousands of incast packets without degrading performance [80, 115]), or dynamically adjusted based on runtime metrics (like throughput, latency, or loss rate). In UBT, receivers dynamically modify the incast factor in response to current loss and timeout events. If the loss rate increases, the factor is reduced to alleviate congestion; conversely, if the loss rate remains low (indicating timely packet reception with no timeouts), the incast factor is increased. Receivers communicate their incast factor, I, by updating the `Incast` field in the OptiReduce header (Figure 7), and the sender then selects the smallest reported value of I for that round.

3.2.3 Minimal Rate Control. Since OptiReduce is resilient to loss, we only require a minimal scheme for rate control to prevent congestion collapse. For that, UBT employs a basic TIMELY-like rate-control mechanism [114], where the sender adjusts flow rates based on RTT feedback derived from timestamps returned by the receiver at regular intervals (every 10th packet) over a separate control channel. If the RTT (or its gradient) is below T_{low}, the sender increases the rate by α, and if the RTT exceeds T_{high}, the rate is reduced by $(1-\beta \cdot (1-T_{high}/\text{RTT}))$. In our experiments, we set $T_{low} = 25\mu$s, $T_{high} = 250\mu$s, $\alpha = 50$ Mbps, and $\beta = 0.5$, when running in a shared environment [114, 153].

Summary: UBT, in conjunction with TAR, *improves tail latency* by minimizing the impact of network congestion; adaptive timeouts bound the latency of the send(bcast)/receive stages, while dynamic incast reduces the number of communication rounds.

3.3 Dispersing Gradient Loss

Finally, to make OptiReduce resilient against drop patterns (e.g., tail drop) in the network, we employ *randomized* Hadamard Transform (HT) [144, 148, 149], which spreads the effect of a dropped gradient over the entire bucket. For example, in Figure 9, HT encodes a bucket (ΔB) and sends it over the network. Upon reception, the last gradient (in red) was lost; however, HT preserves the lost information by slightly perturbing the values of other gradients in the decoded bucket (B). The Mean Squared Error (MSE) between the decoded and received (without HT) bucket, compared to the original one, is 0.01 and 2.53, respectively. That is why, when combined with rotating shard responsibility between nodes (Figure 4), HT lets OptiReduce be more aggressive with the timeout value (t_B) while still reaching high model convergence accuracies (§5).

Summary: HT, together with TAR+UBT, *limits the effect of dropped gradients* by spreading it across the entire bucket, thus preserving the lost information. Additionally, it allows OptiReduce to operate faster, with stringent t_B values, without affecting convergence accuracies (§5).

3.4 Safeguards against Excessive Loss

OptiReduce continuously monitors gradient loss during each AllReduce phase, and if the loss exceeds a predefined threshold, it can either skip the gradient update for that round or automatically halt the training, prompting user intervention. Skipping an update helps minimize potential harm to the overall training process by discarding transient high-loss updates without impacting long-term model accuracy or completion time. This mechanism helps prevent major disruptions in the training process, ensuring users are notified of any accuracy concerns and can make necessary adjustments. Similar techniques are routinely integrated into modern deep-learning pipelines to monitor, track, and recover model accuracy [1, 2, 32, 119].

4 Implementation

We develop OptiReduce as a new collective-communication scheme inside the Gloo library (v0.5.0) [13] and integrate it with PyTorch Distributed (v1.12) [106], a widely used deep-learning framework—allowing OptiReduce to work without modification with a large body of deep-learning models (e.g., CNNs [82, 98, 138], RNNs [56, 85, 113], and Transformers [53, 63, 150]). We pick Gloo due to its simpler design and our familiarity with the codebase; however, we expect OptiReduce will yield similar benefits when operating with other popular libraries (e.g., NCCL [87] or MSCCL [24]).

We extend the C++ implementation of Gloo to support our Transpose AllReduce (TAR) collective and provide support for both reliable transport (over TCP) and our best-effort transport (over UBT). We prototype UBT as a userspace transport

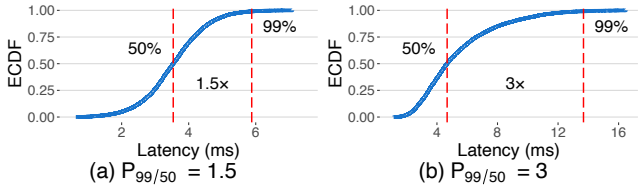


Figure 10: Our local cluster with tail-to-median latency ratio ($P_{99/50}$) of 1.5 (a) and 3 (b).

layered on UDP, including rate control, using Nvidia DPDK API (v20.11) [11].

We further add support for communication hiding in OptiReduce, i.e., running two AllReduce operations in parallel with backpropagation.[7] The sender maintains separate layer-3 port numbers to tag gradients for the two parallel AllReduce operations. On the receive side, two PMD threads poll incoming traffic (gradients) in their local receive queues. An Nvidia Connectx-6 NIC routes traffic to the respective queues based on the port numbers; we install rules in the NIC using DPDK's `rte_flow` API [10]. We also install rules to route non-OptiReduce traffic to the kernel using DPDK's Flow Bifurcation mechanism [9].[8] Doing so ensures that Gloo's kernel stack remains unaffected and other network operations, e.g., rendezvous in PyTorch DDP [106], continue uninterrupted.

To include support for adaptive timeouts, we use C++ STL library's `wait_for()` function [89], which is a blocking call that returns either when a given condition is met (such as received all gradients) or a timeout occurs. For the timeout, we pair the `wait_for()` function with Chrono library's `high_resolution_clock()` [89] to operate at nanosecond clock granularity. For Hadamard Transform, we apply a widely-used C++/CUDA implementation by HazyResearch [15], which uses GPUs to perform this operation. We use PyTorch DDP's communication hook [31] to register Hadamard's encode/decode callbacks for processing gradient buckets before and after reduction, respectively.

5 Evaluation

In this section, we provide an end-to-end comparison of OptiReduce with state-of-the-art solutions (§5.2), and microbenchmark the utility of its design components (§5.3).

5.1 Experimental Setup

5.1.1 Test Environments. We evaluate OptiReduce in both our local virtualized cluster and a real-world environment using CloudLab [66].

a) Local Virtualized Cluster. Our local testbed is a collection of four servers configured as a virtualized cluster [30]. Each machine has a 32-core AMD EPYC 7542 CPU @ 2.90 GHz, 512 GB RAM, two Nvidia Tesla V100 GPUs, and a ConnectX-5 dual-port NIC. In total, there are eight V100 GPUs, one per VM in the cluster. The VMs communicate over the network using a dedicated NIC port with Nvidia's OFED device drivers (v24.04). Both GPU and NIC interfaces are exposed to the VMs via Intel's VT-d PCIe passthrough technology [41]—allowing direct (dedicated) access to the physical functions. A programmable switch (Tofino1 [18]) connects the servers and VMs over a 25 Gbps network. Additionally, it facilitates in-network aggregation for SwitchML benchmarks (§5.3).

Recent studies from Microsoft [137], Amazon [91], and Google [61, 62] show that the tail-to-median ratio ($P_{99/50}$) for distributed workloads, including deep-learning training, ranges from 1.5× to 4× in large cloud data centers [40, 61, 106].[9] To emulate these environments and their tail characteristics in our testbed, we follow the approach of previous studies [42,45,46,151,165] by running background workloads on random nodes and links. Varying the number of concurrent workloads allows us to adjust the tail-to-median latency ratio within the network. We validate the fidelity of our scheme using the Gloo benchmark utility [12] with 2K gradients. As shown in Figure 10, our method accurately preserves the expected latency distributions, maintaining the $P_{99/50} = 1.5, 3$ ratios.

b) Public Cloud: CloudLab. We configured our real environment on CloudLab [66], a public research cloud widely shared by researchers and academics for computing and distributed systems experimentation. We provisioned eight `d7525` instances [8], each equipped with an Nvidia Ampere A30 GPU and a ConnectX-6 DX dual-port NIC, all connected via a 10 Gbps network.

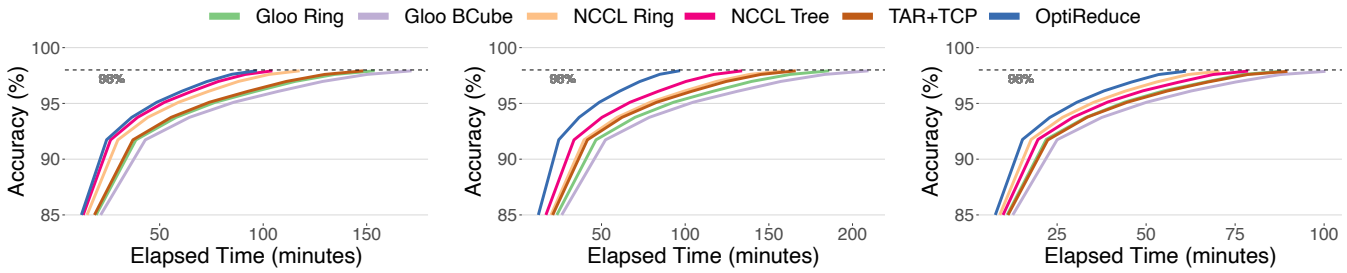
5.1.2 Baselines, Workloads, and Parameter Settings. We evaluate OptiReduce against the following baseline systems: Gloo (Ring [121] and BCube [76]), NCCL (Ring [121] and Tree [22]) with TCP, as well as a reliable version of our Transpose AllReduce (TAR) with TCP (TAR+TCP). Additionally, we evaluate BytePS and three popular compression algorithms: Top-K [142], TernGrad [155], and THC [103]. To provide further insights, we also microbenchmark OptiReduce against in-network systems such as SwitchML [136], despite their reliance on switch-level access within the provider's network, which makes them inapplicable for cloud environments.

We train a variety of language models (LMs), including BERT-base/large [63] and RoBERTa-base/large [111] on the SQuAD 2.0 dataset [129], as well as BART-base/large [101] and OpenAI GPT-2-base/large [126] on the GLUE benchmark [152] for the SST2 (Stanford Sentiment Treebank) task [139]. We further evaluate OptiReduce on additional models and tasks, which we discuss in Appendix B and C. Specifically, we train the Llama-3.2 1B model [65] on three

[7] This is consistent with existing parallelism approaches that allow for two concurrent AllReduce operations (e.g., PyTorch [106]).

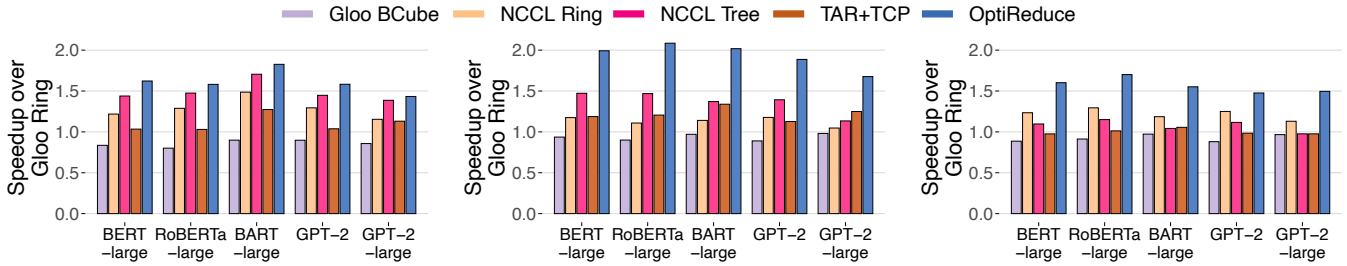
[8] Flow Bifurcation is a mechanism that lets hardware-capable NICs forward traffic directly to the userspace (DPDK thread) or the Linux kernel.

[9] Even CloudLab, a relatively small-scale cloud compared to commercial ones, exhibits a $P_{99/50}$ ratio of around 1.45.



(a) Local Cluster: $P_{99/50} = 1.5$ **(b)** Local Cluster: $P_{99/50} = 3$ **(c)** CloudLab

Figure 11: Time-to-accuracy (TTA) comparison for the OpenAI GPT-2 model with eight worker nodes.



(a) Local Cluster: $P_{99/50} = 1.5$ **(b)** Local Cluster: $P_{99/50} = 3$ **(c)** CloudLab

Figure 12: Training throughput comparison for large language models (LLMs) with eight worker nodes.

standard downstream tasks: SQuAD (extractive question answering) [129], ARC (science reasoning) [58], and MATH (symbolic mathematics) [83] (Appendix B). Additionally, we evaluate and microbenchmark OptiReduce on network-intensive models (VGG-16/19) [138] using the CIFAR-100 dataset [97] and compute-intensive models (ResNet-50/101/152) [82] with the ImageNet dataset [135] (Appendix C).

We compute the OptiReduce's timeout value (t_B) for each model using 20 iterations; we set $\alpha = 0.95$ when calculating the moving average (t_C). We use the incast parameter of $I = 1$, unless stated otherwise.

5.2 End-to-End Evaluation

We conduct end-to-end evaluations in two environments: (1) our local virtualized cluster, with tail-to-median ratios $P_{99/50} = 1.5$ (low variability) and 3 (high variability), and (2) a real public cloud, CloudLab. We compare OptiReduce against the baseline systems Gloo (Ring and BCube), NCCL (Ring and Tree), and TAR+TCP; and measure time-to-accuracy (TTA), throughput, gradient drop percentage (in bytes), and the achieved training accuracy.

Our results show that OptiReduce consistently outperforms the baselines. On our local cluster, we observe time-to-accuracy (TTA) reductions of up to (82%, 98%) compared to Gloo (Ring, BCube), and (44%, 25%) compared to NCCL (Ring, Tree), respectively. These improvements extend to CloudLab, where we see average TTA reductions of up to (47%, 67%) over Gloo (Ring, BCube), and (18%, 32%) over NCCL (Ring, Tree). Furthermore, OptiReduce achieves the same convergence accuracy as the baselines while limiting gradient entry losses to less than 0.1% of the total traffic.

• **TTA and Throughput.** Figure 11 illustrates how TTA for the five baselines and OptiReduce varies under different environments—Local cluster ($P_{99/50}$ = 1.5 and 3) and CloudLab—for the OpenAI GPT-2 model. Across all runs, OptiReduce maintains a lower TTA from the onset.[10] For example, on our local cluster with $P_{99/50} = 1.5$ (Figure 11a), OptiReduce converges in 96 minutes, while NCCL Tree takes 105 minutes, and the next best, NCCL Ring, taking 118 minutes. With $P_{99/50} = 3$, the TTA differences become more pronounced (Figure 11b). OptiReduce remains unaffected by the increased variability, maintaining its lead in TTA with a 98% accuracy and finishing in about 97 minutes. In contrast, the baselines experience significant slowdowns, with their TTA inflating by 1.41–2.18× compared to OptiReduce.

We see the same trend on CloudLab (Figure 11c), OptiReduce reaches the convergence accuracy in 60 minutes, whereas it is 71 minutes for NCCL Ring. Other baselines continue to trail behind OptiReduce, with NCCL Tree having the next-best TTA of 79 minutes.

We observe similar speedups for OptiReduce when training other models, including BERT-large, RoBERTa-large, BART-large, and GPT-2-large (Figure 12).

• **Gradient Drops and Convergence Time.** We further evaluate the drops in gradient entries and their impact on convergence time (Table 1). In our local cluster with $P_{99/50} = 1.5$, a small percentage of gradient entries is lost (i.e., 0.07%) due to OptiReduce's adaptive timeouts in UBT, causing the system to progress without waiting on stragglers. These timeouts manifest as dropped gradients in OptiReduce; whereas

[10] We observe that under ideal conditions, with $P_{99/50} = 1$ (no variability), all systems perform similarly (not shown).

Test Environment	Gloo		NCCL		TAR+TCP	OptiReduce:	*Dropped Gradients (%Entries)*
	Ring	*BCube*	*Ring*	*Tree*			
Local Cluster: $P_{99/50} = 1.5$	154	172	118	105	148	96	0.07
Local Cluster: $P_{99/50} = 3.0$	186	210	159	135	166	97	0.18
CloudLab	88	100	71	79	90	60	0.05

Table 1: Comparing the end-to-end convergence time (in minutes) of baseline systems vs. OptiReduce for OpenAI GPT-2 (total gradients, 40 TB). TAR+UDP suffers excessive drops, losing up to 30% of gradients, and fails to converge.

baseline systems stall on these stragglers. Still, OptiReduce achieves the same convergence accuracy (98% for GPT-2) as the baselines but in under 96 minutes, compared to 105 minutes for the next best, NCCL Tree. When $P_{99/50}$ increases to 3, increased congestion in the network and stragglers cause more gradient entries to be lost, but only slightly (0.18%), and does not impact OptiReduce's training accuracy and convergence time, whereas it inflates NCCL Tree's time to 135 minutes.

Similarly, in CloudLab, OptiReduce sees a 0.05% drop in gradient entries, which allows it to reach the convergence accuracy in 60 minutes, compared to its next best, NCCL Ring, taking 18% longer.

5.3 Microbenchmarks

We now evaluate the effectiveness of the individual design components in OptiReduce. We run the VGG-19 model on the CIFAR-100 dataset for these measurements using our local cluster.

• **OptiReduce's TAR topology leads to minimum dropped gradients when using a best-effort transport.** We compare the number of gradients lost across different AllReduce topologies using our Unreliable Bounded Transport (UBT). We measure Mean Squared Error (MSE) to gauge the difference between the original gradients and those received over these topologies for three different schemes on our local cluster with $P_{99/50} = 1.5$: Ring topology in Ring-AllReduce, P2P in PS, and P2P with rounds in TAR, using a 500 M tensor. Ring-AllReduce has the worst MSE (14.55)—an order of magnitude greater than TAR (2.47). The presence of fixed node pairs in Ring-AllReduce (§3.1) propagates losses, resulting in a higher deviation from the original gradients. PS also has a high MSE (9.92) due to excessive incast when all nodes send gradients to the parameter server (PS) simultaneously. In contrast, TAR avoids this by distributing P2P communication over multiple rounds.

• **UBT's dynamic incast improves OptiReduce's latency without overloading the receiver nodes.** We measure the effects of UBT's dynamic incast feature on OptiReduce's training latency. Figure 13 compares two configurations: one where we fix $I = 1$, and the other with dynamically managed incast. The results show that OptiReduce's senders can leverage buffer occupancy at receivers to increase I, thus reducing average latency by about 21% compared to always

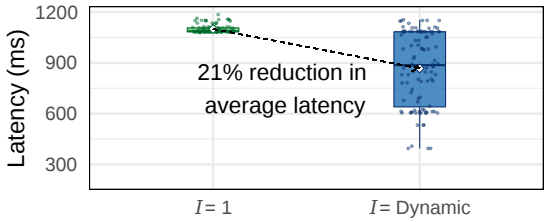
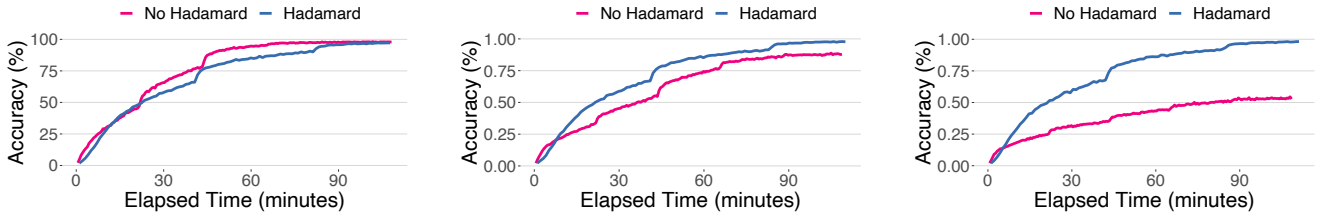


Figure 13: Latency distribution of OptiReduce with static ($I = 1$) vs. dynamic incast feature in UBT, using a synthetic 500 M-gradient AllReduce workload.

sending to a single receiver. This ability to dynamically control the incast parameter (I) allows OptiReduce to adapt itself based on the capacity of the receivers' resources, which is not the case with PS (all workers send to parameter server) or Ring-AllReduce (a receiver interacts with a single sender).

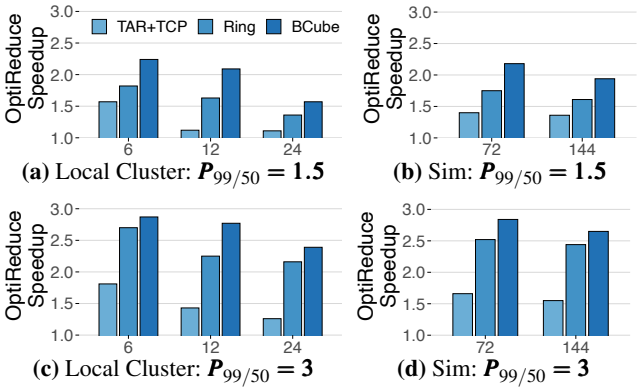
• **OptiReduce's early timeout strategy enables faster progress towards TTA.** We evaluate the effectiveness of the early timeout strategy (t_C) in OptiReduce. We disable t_C and only keep the timeout value t_B and measure its effect on training accuracy, time, and dropped gradients. We find that when training VGG-19 with $P_{99/50} = 1.5$, OptiReduce takes 130 minutes to reach convergence accuracy in 200 epochs while incurring 0.02% of gradient drops. Enabling early timeout brings this training time down by about 16% (to 112 minutes) with a similar drop rate (0.02%). By adapting t_C, OptiReduce sustains the same drop rate and finishes quickly, rather than waiting for the higher t_B value each time. We notice that with early timeout enabled, OptiReduce triggers t_C 95% more often than t_B; hence, resulting in faster TTAs.

• **OptiReduce's Hadamard Transform (HT) allows it to reach convergence accuracies even under higher percentages of dropped gradients.** Figure 14 shows the training accuracy of VGG-19 model with and without Hadamard enabled. When considering TTA, we see that Hadamard introduces some computational overhead when operating with only 1% of dropped gradient entries (Figure 14a). It takes Hadamard more time to reach convergence accuracies (around 97 minutes) compared to when it is disabled (90 minutes). However, as drops increase (5% or more), it starts to outperform the non-Hadamard instance with much faster TTAs (Figure 14b,c). Looking closely, we notice that across all dropped percentages, Hadamard is able to sustain the same TTA ($\approx$ 97 minutes)—showing its resilience to drops. In contrast, the non-Hadamard case quickly degrades and fails to achieve



(a) 1% gradient drops **(b)** 5% gradient drops **(c)** 10% gradient drops

Figure 14: Training accuracy of VGG-19 with/without Hadamard Transform (HT) in OPTIREDUCE.



(a) Local Cluster: $P_{99/50} = 1.5$ **(b)** Sim: $P_{99/50} = 1.5$

(c) Local Cluster: $P_{99/50} = 3$ **(d)** Sim: $P_{99/50} = 3$

Figure 15: OPTIREDUCE speedup over baseline systems (TAR+TCP, Ring, BCube) with increasing #workers using a synthetic 500 M-gradient AllReduce workload.

convergence accuracy even under 10% drops. The percentage drops include both drops incurred due to network variabilities (e.g., congestion and retransmissions) and gradients that a slow worker could not send due to timeouts.

• **OPTIREDUCE scales with increasing number of worker nodes, consistently maintaining high speedups.** To demonstrate OPTIREDUCE's performance at scale, we first run tests using CPU-based worker nodes on our local cluster. We compare OPTIREDUCE with TAR+TCP and Gloo (Ring and BCube) on a synthetic AllReduce workload, aggregating 500 M gradient entries across 6–24 nodes (Figure 15a, c).[11] Next, we conduct simulations with larger clusters (72 and 144 nodes), similar in sizes to prior works [43, 100, 107, 153]—using latencies sampled from the local cluster and scaled for higher node counts (Figure 15b, d). Across all tests, OPTIREDUCE consistently delivers high speedups, achieving 2× improvements over Ring and BCube in high-tail environments, $P_{99/50} = 3$ (Figure 15c, d).

• **Unlike OPTIREDUCE, lossy/compression schemes are vulnerable to tail effects in shared environments.** Though OPTIREDUCE is orthogonal to sparsification and quantization techniques, our comparison (Figure 16) shows that lossy/compression schemes (e.g., Top-K, TernGrad, and THC) fail to effectively address tail effects. While these schemes reduce the volume of gradient entries shared, they rely on a static evaluation of how much data to compress (or drop) a priori before transmission. In contrast, OPTIREDUCE handles loss

[11] We exclude NCCL from this comparison as it relies on GPUs.

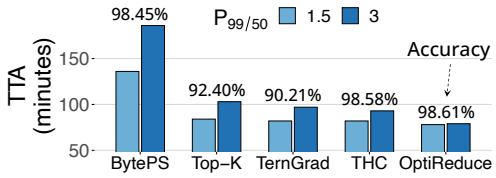


Figure 16: OPTIREDUCE comparison with lossy/compression schemes (BytePS, Top-K, TernGrad, & THC), showing TTA and their convergence accuracy.

in real-time, dynamically adapting to network conditions and minimizing tail latency. For instance, THC matches OPTIREDUCE in convergence accuracy but takes 4% and 18% longer to complete under $P_{99/50} = 1.5$ and 3, respectively. Other schemes perform even worse, either requiring 2× more time to converge or stalling at lower accuracies due to their lossy compression, failing to improve end-to-end TTA even with additional training epochs [103, 153].

• **In-network aggregation (INA) approaches struggle with tail effects, while OPTIREDUCE remains unaffected.** In-network aggregation (INA) methods, such as SwitchML [136], reduce network latency through hardware-accelerated aggregation within the network. However, they remain vulnerable to tail effects—significantly inflating their completion times as the tail-to-median ratio increases. For instance, in a low-tail environment ($P_{99/50} = 1.5$), SwitchML performs 52% faster than OPTIREDUCE. However, as the tail-to-median ratio increases from $P_{99/50} = 1.5$ to 3, its completion time rises by about 2.1×, even surpassing OPTIREDUCE by 28%. In contrast, OPTIREDUCE remains unaffected by this change while reaching convergence. By bypassing stragglers and proceeding without waiting for all gradients, OPTIREDUCE is better suited for shared and high-tail environments. Moreover, OPTIREDUCE's design can be extended to incorporate INA, potentially achieving similar speedups in low-tail environments—an avenue we plan to explore in future work.

6 Limitations & Future Work

In the current AllReduce design, there are two primary bottlenecks: (a) in the computation (or reduction) phase and (b) in the communication phase. We explore potential solutions for both bottlenecks in the subsequent sections.

a) Accelerators for Reduction. In OPTIREDUCE, we primarily focus on bounding the execution time of the two send/receive stages in AllReduce (Figure 1). The reduction stage, i.e.,

the process of averaging gradients together, still happens on CPUs. However, as models grow and gradient sizes increase, the reduction stage can become a bottleneck. Rather than opting for the most extreme case of offloading all of AllReduce to network switches [99,136], we can instead consider offloading the reduction operation on the end-host server (similar to how we accelerate GEMMs using GPUs) [141]. Modern SmartNICs [19,25,26], with onboard FPGAs and ML accelerators, can present a promising opportunity for accelerating reduction. But, doing so requires rethinking and redesigning the application interface (API) between OptiReduce and SmartNICs. We hope OptiReduce to serve as a stepping stone for research in this direction.

b) Accelerators for Network Transport. As with reduction, network transport can also become a bottleneck with link/interface speeds reaching 400 Gbps+. Existing offloads, like RDMA [77], provide high-bandwidth communication between nodes by moving data to/from the main memory and the network without engaging the host CPU. However, these implementations are still susceptible to tail effects in the network (e.g., packet drops and retransmissions). We hope OptiReduce's transport design can offer guidance in building new offloads for network transport, particularly with support for unreliable bounded protocols. As a next step, we could explore offloading OptiReduce's transport onto hardware using RDMA's Unreliable Connected (UC) or Unreliable Datagram (UD) queue pairs [49]. However, these implementations currently suffer from excessive packet drops when packets arrive out of order [23]. We plan to address these challenges in future work.

7 Related Work

Lossy Architectures for Accelerating Allreduce Collectives. THC [103] presents compression-aware gradient synchronization architectures for DNN training, introducing homomorphic compression to reduce bandwidth through quantization. OmniReduce [68] introduces the concept of a streaming aggregation, which exploits parameter sparsity to maximize effective bandwidth use by sending only non-zero data blocks. MLT [153] configures network switches to prioritize and drop packets based on model layers and gradient magnitudes, leveraging inter-packet order-independency to balance load. In contrast, OptiReduce exploits DDL's resiliency to gradient drops in mitigating tail effects while sustaining convergence accuracies in the cloud without requiring access to the provider's network. It could apply techniques like OmniReduce to reduce network usage for models with sparse gradients or use quantization methods similar to THC.

Accelerating Deep Learning using In-Network Computing. SHArP [75] is a hierarchical aggregation protocol and architecture in Nvidia Switches (e.g., IB-2 [35]), which builds an overlay reduction tree for aggregating data flowing through the switch. SwitchML [136] accelerates distributed training by using a programmable data-plane device (e.g., Intel Tofino [18,51]) to aggregate the model updates from multiple workers in the network. To overcome the switch memory and computational constraints, they co-design the in-switch processing with end-host protocols (e.g., sliding window of parameters) for handling drops. ATP [99] is an in-network aggregation solution similar to SwitchML, but for deep learning, and is designed to provide a dynamic, best-effort in-network aggregation service for multi-tenant multi-switch clusters. Unlike these solutions, OptiReduce does not require specialized hardware or access to the provider's network.

Optimizing Deep Learning for Enterprise and HPC Environments. Cassini [128] is a network-aware job scheduler for ML clusters in HPC environments that optimizes network resource usage by interleaving communication patterns of ML jobs, reducing congestion and improving cluster performance. Meta's recent paper [72] presents a custom backend for distributed deep-learning training targeting enterprise data centers. It optimizes network topology, job scheduling, placement, and data transport to improve training performance, efficiency, and scalability. On the other hand, OptiReduce offers a resilient and tail-optimal solution for deep-learning training in the cloud.

8 Conclusion

OptiReduce leverages distributed-deep learning's (DDL) resiliency to lost gradients and achieves speedups of up to (70%, 30%), on average, over existing frameworks (Gloo, NCCL), in shared environments (e.g., public clouds). OptiReduce implements a domain-specific Transpose Allreduce collective algorithm with unreliable bounded transport (UBT) featuring adaptive timeouts, while mitigating the impact of gradient loss using Hadamard Transform. It delivers higher tail performance (e.g., TTA and training throughput) while preserving DDL models' convergence accuracy and limiting gradient drops to under 0.1%. Looking forward, we hope OptiReduce inspires further exploration of the tradeoff between tail performance and training accuracy in processing contemporary deep-learning models.

Acknowledgements

We sincerely appreciate the guidance of our shepherd, Changhoon Kim, as well as Ashwin Murthy, Roop Mukherjee, Leo Liu, Minlan Yu, Tushar Krishna, Ajay Brahmakshatriya, and the anonymous reviewers for their valuable feedback in strengthening this paper. We also thank Roop Mukherjee, Leo Liu, Ali Imran, and Ali Aqdas for helping with the artifact and initial background studies on tail behavior in distributed training environments, conducted on Nvidia's internal shared clusters and AWS EC2 instances. This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA; by NSF awards CAREER-2338034 and CNS-2211381; and through a Google Research Scholar Award. Support also came in part via a generous gift from Nvidia.

References

[1] Collaborative AI Platform. https://www.evidentlyai.com. Last accessed: 01/2025.

[2] neptune.ai AI Experiment Tracker. https://neptune.ai. Last accessed: 01/2025.

[3] Amazon AWS EC2. https://aws.amazon.com/ec2. Last accessed: 01/2025.

[4] Amazon AWS EC2 Elastic Fabric Adapter (EFA). https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/efa.html. Last accessed: 01/2025.

[5] Amazon EC2 P3 Instance. https://aws.amazon.com/ec2/instance-types/p3. Last accessed: 01/2025.

[6] Amazon EC2 P4 Instance. https://aws.amazon.com/ec2/instance-types/p4. Last accessed: 01/2025.

[7] Amazon EC2 P5 Instance. https://aws.amazon.com/ec2/instance-types/p5. Last accessed: 01/2025.

[8] CloudLab Hardware. https://docs.cloudlab.us/hardware.html. Last accessed: 08/2024.

[9] DPDK Flow Bifurcation Guide. https://doc.dpdk.org/guides-20.11/howto/flow_bifurcation.html. Last accessed: 01/2025.

[10] DPDK Generic Flow API Docs. https://doc.dpdk.org/guides-20.11/prog_guide/rte_flow.html. Last accessed: 01/2025.

[11] DPDK v20.11. https://doc.dpdk.org/guides-20.11. Last accessed: 01/2025.

[12] Gloo Benchmarking Utility. https://github.com/facebookincubator/gloo?tab=readme-ov-file#benchmark. Last accessed: 01/2025.

[13] Gloo Collective-Communication Library. https://github.com/facebookincubator/gloo. Last accessed: 01/2025.

[14] Grand View Research: Cloud AI Market Report. https://www.grandviewresearch.com/industry-analysis/cloud-ai-market-report. Last accessed: 01/2025.

[15] HazyResearch's Open Source Hadamard CUDA Implementation. https://github.com/HazyResearch/structured-nets. Last accessed: 01/2025.

[16] How Cloud Computing Revolutionized Business Operations. https://www.forbes.com/sites/emilsayegh/2023/11/28/how-cloud-computing-revolutionized-business-operations-and-what-lies-ahead. Last accessed: 01/2025.

[17] Hyperstack: Complete Guide to Machine Learning for Cloud 2024. https://www.hyperstack.cloud/blog/case-study/complete-guide-to-machine-learning-for-cloud-2024. Last accessed: 01/2025.

[18] Intel Tofino P4 Switch. https://www.intel.com/content/www/us/en/products/details/network-io/programmable-ethernet-switch/tofino-series.html. Last accessed: 01/2025.

[19] Intel® FPGA SmartNIC. https://www.intel.com/content/www/us/en/products/details/fpga/platforms/smartnic.html. Last accessed: 01/2025.

[20] Machine Learning in the Cloud. https://www.run.ai/guides/machine-learning-in-the-cloud. Last accessed: 01/2025.

[21] Machine Learning in the Cloud: What Are the Benefits. https://symphony-solutions.com/insights/machine-learning-in-the-cloud-what-are-the-benefits. Last accessed: 01/2025.

[22] Massively Scale Your Deep Learning Training with NCCL 2.4. https://developer.nvidia.com/blog/massively-scale-deep-learning-training-nccl-2-4. Last accessed: 01/2025.

[23] Mellanox Support for Out-of-order Packets Data Placement. https://docs.nvidia.com/networking/display/mlnxofedv543100/out-of-order+(ooo)+data+placement. Last accessed: 01/2025.

[24] Microsoft Collective-Communication Library. https://github.com/microsoft/msccl. Last accessed: 01/2025.

[25] napa:tech; SmartNICs. https://www.napatech.com/products. Last accessed: 01/2025.

[26] NVIDIA ConnectX Series SmartNIC. https://www.nvidia.com/en-us/networking/ethernet-adapters. Last accessed: 01/2025.

[27] OpenAI's GPT-3 Language Model: A Technical Overview. https://lambdalabs.com/blog/demystifying-gpt-3. Last accessed: 01/2025.

[28] Optimizing Artificial Intelligence and Machine Learning with Cloud Computing. https://www.orange-business.com/en/blogs/optimizing-artificial-intelligence-machine-learning-cloud-computing. Last accessed: 01/2025.

[29] OptiReduce Public Artifact. https://github.com/OptiReduce.

[30] Proxmox VM Environment. https://pve.proxmox.com/wiki/Main_Page. Last accessed: 01/2025.

[31] PyTorch DDP Communication Hook. https://pytorch.org/docs/stable/ddp_comm_hooks.html. Last accessed: 01/2025.

[32] PyTorch TorchFT: Per-step Fault Tolerance. https://github.com/pytorch/torchft. Last accessed: 01/2025.

[33] Redress Compliance: Exploring Cloud-Based Machine Learning Platforms. https://redresscompliance.com/exploring-cloud-based-machine-learning-platforms. Last accessed: 01/2025.

[34] RunPod - The Cloud Built for AI. https://www.runpod.io. Last accessed: 01/2025.

[35] Switch-IB™ 2 EDR. https://network.nvidia.com/files/doc-2020/pb-switchib2-edr-switch-silicon.pdf. Last accessed: 01/2025.

[36] The Power of Machine Learning in the Cloud: Transforming Business Operations. https://mentormate.com/blog/the-power-of-machine-learning-in-the-cloud-transforming-business-operations. Last accessed: 01/2025.

[37] The Rise of Cloud Machine Learning. https://encapture.com/the-rise-of-cloud-machine-learning. Last accessed: 01/2025.

[38] Criteo: Terabyte Click Logs Dataset. https://labs.criteo.com/2013/12/download-terabyte-click-logs, 2013.

[39] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale Machine Learning on Heterogeneous Distributed Systems. *arXiv:1603.04467*, 2016.

[40] Sepehr Abbasi, Shiva Ketabi, Ali Munir, Mahmoud Bahnasy, and Yashar Ganjali. DWTCP: Ultra Low Latency Congestion Control Protocol for Data Centers. *arXiv:2207.05624*, 2022.

[41] Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and John Wiegert. Intel Virtualization Technology for Directed I/O. *Intel technology journal*, 10(3), 2006.

[42] Vamsi Addanki, Oliver Michel, and Stefan Schmid. PowerTCP: Pushing the Performance Limits of Datacenter Networks. In *USENIX NSDI*, 2022.

[43] Saksham Agarwal, Qizhe Cai, Rachit Agarwal, David Shmoys, and Amin Vahdat. Harmony: A Congestion-free Datacenter Architecture. In *USENIX NSDI*, 2024.

[44] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. QSGD: Communication-efficient SGD via Gradient Quantization and Encoding. *NeurIPS*, 2017.

[45] Mohammad Alizadeh and Tom Edsall. On the Data Path Performance of Leaf-spine Datacenter Fabrics. In *High-Performance Interconnects*. IEEE, 2013.

[46] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *ACM SIGCOMM*, 2014.

[47] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *ACM SIGCOMM*, 2010.

[48] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective Straggler Mitigation: Attack of the Clones. In *USENIX NSDI*, 2013.

[49] Dotan Barak. RDMA Aware Networks Programming User Manual. *NVIDIA Docs*, 2015.

[50] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, Shyamal Buch, Dallas Card, Rodrigo Castellon, Niladri Chatterji, Annie Chen, Kathleen Creel, Jared Quincy Davis, Dora Demszky, Chris Donahue, Moussa Doumbouya, Esin Durmus, Stefano Ermon, John Etchemendy, Kawin Ethayarajh, Li Fei-Fei, Chelsea Finn, Trevor Gale, Lauren Gillespie, Karan Goel, Noah Goodman, Shelby Grossman, Neel Guha, Tatsunori Hashimoto, Peter Henderson, John Hewitt, Daniel E. Ho, Jenny Hong, Kyle Hsu, Jing Huang, Thomas Icard, Saahil Jain, Dan Jurafsky, Pratyusha Kalluri, Siddharth Karamcheti, Geoff Keeling, Fereshte Khani, Omar Khattab, Pang Wei Koh, Mark Krass, Ranjay Krishna, Rohith Kuditipudi, Ananya Kumar, Faisal Ladhak, Mina Lee, Tony Lee, Jure Leskovec, Isabelle Levent, Xiang Lisa Li, Xuechen Li, Tengyu Ma, Ali Malik, Christopher D. Manning, Suvir Mirchandani, Eric Mitchell, Zanele Munyikwa, Suraj Nair, Avanika Narayan, Deepak Narayanan, Ben Newman, Allen Nie, Juan Carlos Niebles, Hamed Nilforoshan, Julian Nyarko, Giray Ogut, Laurel Orr, Isabel Papadimitriou, Joon Sung Park, Chris Piech, Eva Portelance, Christopher Potts, Aditi Raghunathan, Rob Reich, Hongyu Ren, Frieda Rong, Yusuf Roohani, Camilo Ruiz, Jack Ryan, Christopher Ré, Dorsa Sadigh, Shiori Sagawa, Keshav Santhanam, Andy Shih, Krishnan Srinivasan, Alex Tamkin, Rohan Taori, Armin W. Thomas, Florian Tramèr, Rose E. Wang, William Wang, Bohan Wu, Jiajun Wu, Yuhuai Wu, Sang Michael Xie, Michihiro Yasunaga, Jiaxuan You, Matei Zaharia, Michael Zhang, Tianyi Zhang, Xikun Zhang, Yuhui Zhang, Lucia Zheng, Kaitlyn Zhou, and Percy Liang. On the Opportunities and Risks of Foundation Models. *arXiv:2108.07258*, 2021.

[51] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. *ACM SIGCOMM CCR*, 2013.

[52] Léon Bottou. Stochastic gradient descent tricks. *Neural Networks: Tricks of the Trade: Second Edition*, 2012.

[53] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models Are Few-shot Learners. *NeurIPS*, 2020.

[54] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting Distributed Synchronous SGD. *arXiv:1604.00981*, 2016.

[55] 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:1512.01274*, 2015.

[56] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation. *arXiv:1406.1078*, 2014.

[57] James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory R Ganger, Garth Gibson, Kimberly Keeton, and Eric P Xing. Solving the straggler problem with bounded staleness. *White paper, Carnegie Mellon University*, 2013.

[58] Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think You Have Solved Question Answering? Try ARC, The AI2 Reasoning Challenge. *arXiv:1803.05457*, 2018.

[59] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. The MPI Message Passing Interface Standard. In *Programming Environments for Massively Parallel Distributed Systems, Springer*, 1994.

[60] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A Low-Latency Online Prediction Serving System. In *USENIX NSDI*, 2017.

[61] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Communications of the ACM*, 56(2), 2013.

[62] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. Large Scale Distributed Deep Networks. *NeurIPS*, 2012.

[63] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv:1810.04805*, 2018.

[64] Xianzhi Du, Mostafa El-Khamy, Jungwon Lee, and Larry Davis. Fused DNN: A Deep Neural Network Fusion Approach to Fast and Robust Pedestrian Detection. In *IEEE winter conference on applications of computer vision (WACV)*, 2017.

[65] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The Llama 3 Herd of Models. *arXiv:2407.21783*, 2024.

[66] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The Design and Operation of CloudLab. In *USENIX ATC*, 2019.

[67] Sanghamitra Dutta, Gauri Joshi, Soumyadip Ghosh, Parijat Dube, and Priya Nagpurkar. Slow and Stale Gradients Can Win The Race: Error-runtime Trade-offs in Distributed SGD. In *International conference on artificial intelligence and statistics*, 2018.

[68] Jiawei Fei, Chen-Yu Ho, Atal N Sahu, Marco Canini, and Amedeo Sapio. Efficient Sparse Collective Communication and its Application to Accelerate Distributed Deep Learning. In *ACM SIGCOMM*, 2021.

[69] Xu Fei, Xu Jianian, Chen Jiabin, Chen Li, Shang Ruitao, Zhou Zhi, and Liu Fangming. iGniter: Interference-Aware GPU Resource Provisioning for Predictable DNN Inference in the Cloud. *arXiv:2211.01713*, 2022.

[70] Shuo Feng, Huiyu Zhou, and Hongbiao Dong. Using Deep Neural Network with Small Dataset to Predict Material Defects. *Materials & Design*, 162, 2019.

[71] Mingsheng Fu, Hong Qu, Zhang Yi, Li Lu, and Yongsheng Liu. A Novel Deep Learning-based Collaborative Filtering Model for Recommendation System. *IEEE transactions on cybernetics*, 49(3), 2018.

[72] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, Shuqiang Zhang, Mikel Jimenez Fernandez, Shashidhar Gandham, and Hongyi Zeng. RDMA over Ethernet for Distributed Training at Meta Scale. In *ACM SIGCOMM*, 2024.

[73] James Salamy Ayush Sharma Manya Ghobadi and Muriel Médard. FlexEnt: Entropy Coding to Curb Stragglers in Large-Scale Distributed Machine Learning. *Workshop on AI Systems at SOSP*, 2019.

[74] Abhinav Goel, Caleb Tung, Yung-Hsiang Lu, and George K Thiruvathukal. A Survey of Methods for Low-power Deep Learning and Computer Vision. In *IEEE 6th World Forum on Internet of Things (WF-IoT)*, 2020.

[75] Richard L. Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, Lion Levi, Alex Margolin, Tamir Ronen, Alexander Shpiner, Oded Wertheim, and Eitan Zahavi. Scalable Hierarchical Aggregation Protocol (SHArP): A Hardware Architecture for Efficient Data Reduction. In *IEEE International Workshop on Communication Optimizations in HPC (COMHPC)*, 2016.

[76] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. BCube: A High Performance, Server-Centric Network Architecture for Modular Data Centers. In *ACM SIGCOMM*, 2009.

[77] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over Commodity Ethernet at Scale. In *ACM SIGCOMM*, 2016.

[78] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A Large-scale System for Data Center Network Latency Measurement and Analysis. In *ACM SIGCOMM*, 2015.

[79] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep Learning with Limited Numerical Precision. In *ICML*, 2015.

[80] 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 *ACM SIGCOMM*, 2017.

[81] Aaron Harlap, Henggang Cui, Wei Dai, Jinliang Wei, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, and Eric P Xing. Addressing the Straggler Problem for Iterative Xonvergent Parallel ML. In *ACM Symposium on Cloud Computing*, 2016.

[82] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *IEEE CCVPR*, 2016.

[83] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring Mathematical Problem Solving with the MATH Dataset. *NeurIPS*, 2021.

[84] Brian Heredia, Joseph D Prusa, and Taghi M Khoshgoftaar. Social Media for Polling and Predicting United States Election Outcome. *Social Network Analysis and Mining*, 8(1), 2018.

[85] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural computation*, 9(8), 1997.

[86] Zhenhua Huang, Guangxu Shan, Jiujun Cheng, and Jian Sun. TRec: An Efficient Recommendation System for Hunting Passengers with Deep Neural Networks. *Neural Computing and Applications*, 31(1), 2019.

[87] Sylvain Jeaugey. NCCL 2.0. In *GPU Technology Conference (GTC)*, 2017.

[88] 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 *USENIX OSDI*, 2020.

[89] Nicolai M Josuttis. The C++ Standard Library: A Tutorial and Reference. *Addison-Wesley*, 2012.

[90] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *ISCA*, 2017.

[91] Can Karakus, Rahul Huilgol, Fei Wu, Anirudh Subramanian, Cade Daniel, Derya Cavdar, Teng Xu, Haohan Chen, Arash Rahnama, and Luis Quintela. Amazon Sagemaker Model Parallelism: A General and Flexible Framework for Large Model Training. *arXiv:2111.05972*, 2021.

[92] Can Karakus, Yifan Sun, Suhas Diggavi, and Wotao Yin. Straggler Mitigation in Distributed Optimization through Data Encoding. *NeurIPS*, 2017.

[93] Asif Iqbal Khan, Junaid Latief Shah, and Mohammad Mudasir Bhat. CoroNet: A Deep Neural Network for Detection and Diagnosis of COVID-19 from Chest X-Ray Images. *Computer methods and programs in biomedicine*, 196, 2020.

[94] Jack Kiefer and Jacob Wolfowitz. Stochastic Estimation of The Maximum of a Regression Function. *The Annals of Mathematical Statistics*, 1952.

[95] Michael Ko, Daniel Eisenhauer, and Renato Recio. A Case for Convergence Enhanced Ethernet: Requirements and Applications. In *IEEE International Conference on Communications*, 2008.

[96] Michał Koziarski and Bogusław Cyganek. Image Recognition with Deep Neural Networks in Presence of Noise–Dealing with and Taking Advantage of Distortions. *Integrated Computer-Aided Engineering*, 24(4), 2017.

[97] Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images. *White paper, University of Toronto*, 2009.

[98] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet Classification with Deep Convolutional Neural Networks. *Communications of the ACM*, 60(6), 2017.

[99] 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 *USENIX NSDI*, 2021.

[100] Jason Lei and Vishal Shrivastav. Seer: Enabling Future-Aware Online Caching in Networked Systems. In *USENIX NSDI*, 2024.

[101] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. *arXiv:1910.13461*, 2019.

[102] Hao Li, Asim Kadav, Erik Kruus, and Cristian Ungureanu. Malt: Distributed Data-Parallelism for Existing ML Applications. In *European Conference on Computer systems*, 2015.

[103] Minghao Li, Ran Ben Basat, Shay Vargaftik, ChonLam Lao, Kevin Xu, Michael Mitzenmacher, and Minlan Yu. THC: Accelerating Distributed Deep Learning Using Tensor Homomorphic Compression. In *USENIX NSDI*, 2024.

[104] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. Communication Efficient Distributed Machine Learning with the Parameter Server. *NeurIPS*, 27, 2014.

[105] Mu Li, Li Zhou, Zichao Yang, Aaron Li, Fei Xia, David G Andersen, and Alexander Smola. Parameter Server for Distributed Machine Learning. In *Big learning NIPS workshop*, 2013.

[106] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch Distributed: Experiences on Accelerating Data Parallel Training. *arXiv:2006.15704*, 2020.

[107] Wenxin Li, Xin He, Yuan Liu, Keqiu Li, Kai Chen, Zhao Ge, Zewei Guan, Heng Qi, Song Zhang, and Guyue Liu. Flow Scheduling with Imprecise Knowledge. In *USENIX NSDI*, 2024.

[108] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. LossRadar: Fast Detection of Lost Packets in Data Center Networks. In *ACM CoNEXT*, 2016.

[109] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. *arXiv:1712.01887*, 2017.

[110] Xiaodong Liu, Pengcheng He, Weizhu Chen, and Jianfeng Gao. Multi-task Deep Neural Networks for Natural Language Understanding. *arXiv:1901.11504*, 2019.

[111] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A Robustly Optimized BERT Pretraining Approach. *arXiv:1907.11692*, 2019.

[112] Pradeep Kumar Mallick, Seuc Ho Ryu, Sandeep Kumar Satapathy, Shruti Mishra, Gia Nhu Nguyen, and Prayag Tiwari. Brain MRI Image Classification for Cancer Detection using Deep Wavelet Autoencoder-Based Deep Neural Network. *IEEE Access*, 7, 2019.

[113] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. Recurrent Neural Network based Language Model. In *Interspeech*, 2010.

[114] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based Congestion Control for the Datacenter. *ACM SIGCOMM CCR*, 2015.

[115] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A Receiver-driven Low-latency Transport Protocol using Network Priorities. In *ACM SIGCOMM*, 2018.

[116] Philipp Moritz, Robert Nishihara, Ion Stoica, and Michael I Jordan. Sparknet: Training Deep Networks in Spark. *arXiv:1511.06051*, 2015.

[117] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely Dataflow System. In *ACM SOSP*, 2013.

[118] 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 *ACM SOSP*, 2019.

[119] David Nigenda, Zohar Karnin, Muhammad Bilal Zafar, Raghu Ramesha, Alan Tan, Michele Donini, and Krishnaram Kenthapadi. Amazon Sagemaker Model Monitor: A System for Real-time Insights into Deployed Machine Learning Models. In *ACM SIGKDD*, 2022.

[120] Xue Ouyang, Changjian Wang, and Jie Xu. Mitigating Stragglers to Avoid QoS Violation for Time-critical Applications through Dynamic Server Blacklisting. *Future Generation Computer Systems*, 101, 2019.

[121] Pitch Patarasuk and Xin Yuan. Bandwidth Optimal All-reduce Algorithms for Clusters of Workstations. *Journal of Parallel and Distributed Computing*, 69(2), 2009.

[122] Fabrizio Petrini, Darren J Kerbyson, and Scott Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *ACM/IEEE Conference on Supercomputing*, 2003.

[123] Gregory F Pfister. An Introduction to the Infiniband Architecture. *High performance mass storage and parallel I/O*, 42(617-632), 2001.

[124] William K Pratt, Julius Kane, and Harry C Andrews. Hadamard Transform Image Coding. *Proceedings of the IEEE*, 1969.

[125] Yanxing Qi, Yi Guo, and Yuanyuan Wang. Image Quality Enhancement using a Deep Neural Network for Plane Wave Medical Ultrasound Imaging. *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, 2020.

[126] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models Are Unsupervised Multitask Learners. *OpenAI blog*, 2019.

[127] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving datacenter performance and robustness with multipath TCP. *ACM SIGCOMM CCR*, 2011.

[128] Sudarsanan Rajasekaran, Manya Ghobadi, and Aditya Akella. CASSINI: Network-Aware Job Scheduling in Machine Learning Clusters. In *USENIX NSDI*, 2024.

[129] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ Questions for Machine Comprehension of Text. *arXiv:1606.05250*, 2016.

[130] S Ramesh, C Yaashuwanth, K Prathibanandhi, Adam Raja Basha, and T Jayasankar. An Optimized Deep Neural Network based DoS Attack Detection in Wireless Video Sensor Network. *Journal of Ambient Intelligence and Humanized Computing*, 2021.

[131] Cèdric Renggli, Saleh Ashkboos, Mehdi Aghagolzadeh, Dan Alistarh, and Torsten Hoefler. Sparcml: High-performance Sparse Communication for Machine Learning. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019.

[132] Alessandro Rigazzi. DC-S3GD: Delay-Compensated Stale-Synchronous SGD for Large-Scale Decentralized Neural Network Training. In *ACM/IEEE Third Workshop on Deep Learning on Supercomputers (DLS)*, 2019.

[133] Joshua Romero, Junqi Yin, Nouamane Laanait, Bing Xie, M Todd Young, Sean Treichler, Vitalii Starchenko, Albina Borisevich, Alex Sergeev, and Michael Matheson. Accelerating Collective Communication in Data Parallel Training across Deep Learning Frameworks. In *USENIX NSDI*, 2022.

[134] Alexander Rucker, Muhammad Shahbaz, and Kunle Olukotun. Chopping off the Tail: Bounded Non-Determinism for Real-Time Accelerators. *IEEE Computer Architecture Letters (CAL)*, 2021.

[135] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander Berg, and Li Fei-Fei. Imagenet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 2015.

[136] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtárik. Scaling Distributed Machine Learning with In-Network Aggregation. In *USENIX NSDI*, 2021.

[137] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. TACCL: Guiding Collective Algorithm Synthesis using Communication Sketches. In *USENIX NSDI*, 2023.

[138] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-scale Image Recognition. *arXiv:1409.1556*, 2014.

[139] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive Deep Models for Semantic Compositionality Over A Sentiment Treebank. In *Empirical Methods in Natural Language Processing*, 2013.

[140] Akshitha Sriraman, Sihang Liu, Sinan Gunbay, Shan Su, and Thomas F Wenisch. Deconstructing the Tail at Scale Effect Across Network Protocols. *arXiv:1701.03100*, 2017.

[141] Dave Steinkraus, Ian Buck, and PY Simard. Using GPUs for Machine Learning Algorithms. In *IEEE ICDAR*, 2005.

[142] Sebastian U Stich, Jean-Baptiste Cordonnier, and Martin Jaggi. Sparsified SGD with Memory. *NeurIPS*, 2018.

[143] Chen Sun, Abhinav Shrivastava, Saurabh Singh, and Abhinav Gupta. Revisiting Unreasonable Effectiveness of Data in Deep Learning Era. In *IEEE ICCV*, 2017.

[144] Ananda Theertha Suresh, X Yu Felix, Sanjiv Kumar, and H Brendan McMahan. Distributed Mean Estimation with Limited Communication. In *ICML*, 2017.

[145] Rashish Tandon, Qi Lei, Alexandros G Dimakis, and Nikos Karampatziakis. Gradient Coding: Avoiding Stragglers in Distributed Learning. In *ICML*, 2017.

[146] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and Efficient Foundation Language Models. *arXiv:2302.13971*, 2023.

[147] Yuichiro Ueno and Rio Yokota. Exhaustive Study of Hierarchical Allreduce Patterns for Large Messages between GPUs. In *IEEE/ACM CCGRID*, 2019.

[148] Shay Vargaftik, Ran Ben Basat, Amit Portnoy, Gal Mendelson, Yaniv Ben Itzhak, and Michael Mitzenmacher. Eden: Communication-efficient and Robust Distributed Mean Estimation for Federated Learning. In *ICML*, 2022.

[149] Shay Vargaftik, Ran Ben-Basat, Amit Portnoy, Gal Mendelson, Yaniv Ben-Itzhak, and Michael Mitzenmacher. Drive: One-bit Distributed Mean Estimation. *NeurIPS*, 2021.

[150] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All You Need. *NeurIPS*, 2017.

[151] Kashi Venkatesh Vishwanath and Amin Vahdat. Evaluating Distributed Systems: Does Background Traffic Matter? In *USENIX ATC*, 2008.

[152] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. GLUE: A Multi-task Benchmark and Analysis Platform for Natural Language Understanding. *arXiv:1804.07461*, 2018.

[153] Hao Wang, Han Tian, Jingrong Chen, Xinchen Wan, Jiacheng Xia, Gaoxiong Zeng, Wei Bai, Junchen Jiang, Yong Wang, and Kai Chen. Towards Domain-Specific Network Transport for Distributed DNN Training. In *USENIX NSDI*, 2024.

[154] Jianqiao Wangni, Jialei Wang, Ji Liu, and Tong Zhang. Gradient Sparsification for Communication-efficient Distributed Optimization. *arXiv:1710.09854*, 2017.

[155] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Terngrad: Ternary Gradients to Reduce Communication in Distributed Deep Learning. *NeurIPS*, 2017.

[156] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52(4), 2009.

[157] Bruno Missi Xavier, Rafael Silva Guimarães, Giovanni Comarela, and Magnos Martinello. Programmable Switches for In-Networking Classification. In *IEEE INFOCOM*, 2021.

[158] Jiacheng Xia, Gaoxiong Zeng, Junxue Zhang, Weiyan Wang, Wei Bai, Junchen Jiang, and Kai Chen. Rethinking Transport Layer Design for Distributed Machine Learning. In *APNet*, 2019.

[159] Guojun Xiong, Gang Yan, Rahul Singh, and Jian Li. Straggler-resilient Distributed Machine Learning with Dynamic Backup Workers. *arXiv:2102.06280*, 2021.

[160] Zhaoqi Xiong and Noa Zilberman. Do Switches Dream of Machine Learning? Toward In-Network Classification. In *ACM HotNets*, 2019.

[161] Cheng Xu, Duo Chai, Jie He, Xiaotong Zhang, and Shihong Duan. InnoHAR: A Deep Neural Network for Complex Human Activity Recognition. *IEEE Access*, 7, 2019.

[162] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding Long Tails in the Cloud. In *USENIX NSDI*, 2013.

[163] Neeraja J Yadwadkar and Wontae Choi. Proactive Straggler Avoidance using Machine Learning. *White paper, University of Berkeley*, 2012.

[164] Yauhen Yakimenka, Chung-Wei Weng, Hsuan-Yin Lin, Eirik Rosnes, and Jörg Kliewer. Straggler-Resilient Differentially-Private Decentralized Learning. In *IEEE Information Theory Workshop (ITW)*, 2022.

[165] Siyu Yan, Xiaoliang Wang, Xiaolong Zheng, Yinben Xia, Derui Liu, and Weishan Deng. ACC: Automatic ECN Tuning for High-speed Datacenter Networks. In *ACM SIGCOMM*, 2021.

[166] Mingran Yang, Alex Baban, Valery Kugel, Jeff Libby, Scott Mackie, Swamy Sadashivaiah Renu Kananda, Chang-Hong Wu, and Manya Ghobadi. Using Trio: Juniper Networks' Programmable Chipset for Emerging In-network Applications. In *ACM SIGCOMM*, 2022.

[167] Wenpeng Yin, Katharina Kann, Mo Yu, and Hinrich Schütze. Comparative Study of CNN and RNN for Natural Language Processing. *arXiv:1702.01923*, 2017.

[168] Chen Yu, Hanlin Tang, Cedric Renggli, Simon Kassing, Ankit Singla, Dan Alistarh, Ce Zhang, and Ji Liu. Distributed Learning over Unreliable Networks. In *ICML*, 2019.

[169] Xin Yuan, Weite Li, Kui Lin, and Jinglu Hu. A Deep Neural Network Based Hierarchical Multi-Label Classifier for Protein Function Prediction. In *International Conference on Computer, Information and Telecommunication Systems (CITS)*, 2019.

[170] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. In *ACM SIGCOMM*, 2012.

[171] Georgios Zervakis, Hassaan Saadat, Hussam Amrouch, Andreas Gerstlauer, Sri Parameswaran, and Jörg Henkel. Approximate Computing for ML: State-of-the-Art, Challenges and Visions. In *ACM ASPDAC*, 2021.

[172] 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 *USENIX ATC*, 2017.

[173] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. Treadmill: Attributing the Source of Tail Latency through Precise Load Testing and Statistical Inference. *ACM SIGARCH*, 2016.

[174] Zhuo Zhang, Chao Li, Yangyu Tao, Renyu Yang, Hong Tang, and Jie Xu. Fuxi: A Fault-tolerant Resource Management and Job Scheduling System at Internet Scale. In *VLDB Endowment*, 2014.

[175] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex Smola. Parallelized Stochastic Gradient Descent. *NeurIPS*, 2010.

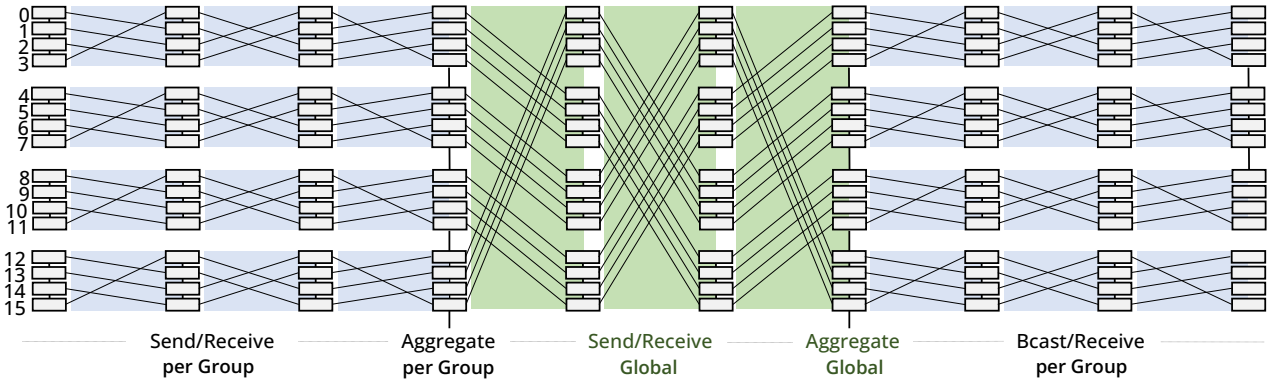


Figure 17: Hierarchical 2D TAR Algorithm.

Environment	Benchmark	Gloo		NCCL		TAR+TCP	OptiReduce:		
		Ring	*BCube*	*Ring*	*Tree*		Time	Accuracy	Test Acc.
$P_{99/50} = 1.5$	ARC	84	113	77	75	76	61	60.45 [+0.45]	39.97 [-0.47]
	MATH	195	254	180	171	175	130	30.56 [+0.18]	30.29 [+0.23]
	SQuAD	4072	5402	3391	3464	3723	3182	46.77 [-0.21]	38.64 [+0.08]
$P_{99/50} = 3.0$	ARC	155	161	128	120	86	61	60.44 [+0.44]	39.91 [-0.53]
	MATH	308	390	299	243	189	131	30.14 [-0.24]	30.09 [+0.03]
	SQuAD	5793	8057	5677	5243	4120	3220	46.45 [-0.53]	38.57 [+0.01]

Table 2: Comparing convergence time (in minutes) and accuracy (% [Δ]), as well as test accuracy (% [Δ]) for the Llama-3.2 1B model across various tasks and environments; [Δ] reports deviation from the baseline accuracy (e.g., Gloo and NCCL).

A Hierarchical 2D TAR Algorithm: Scaling to Larger Node Clusters

In the hierarchical TAR design (Figure 17), nodes are grouped to optimize both intra-group and inter-group communication, reducing the total number of rounds and connections required for AllReduce. For example, with $N = 64$ total nodes divided into $G = 16$ groups, each node communicates only with its corresponding rank across the groups. The number of rounds reduces from $2(N-1) = 126$ in traditional TAR to $2(N/G - 1) + (G-1) = 21$ rounds. The algorithm works in three steps:

- **Intra-group Communication:** Nodes within each group perform send/receive operations followed by aggregation, in parallel, resulting in the *locally* aggregated shard for their rank—taking $(N/G-1)$ rounds.
- **Inter-group Communication:** Corresponding ranks across groups then perform send/receive operations, followed by aggregation, to get the *globally* aggregated shard for their rank—adding another $(G-1)$ rounds.
- **Broadcast Phase:** Finally, nodes within the group broadcast their aggregated shards, which are concatenated to form the globally aggregated gradient bucket—an additional $(N/G-1)$ rounds.

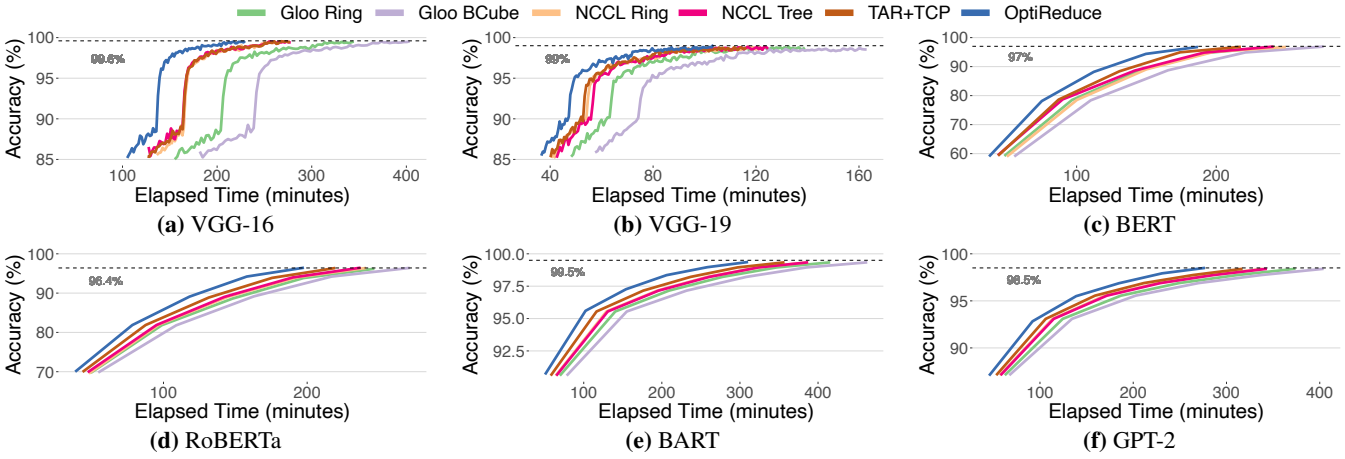
This hierarchical design significantly reduces communication overhead, improving scalability and efficiency for large-scale distributed training.

B Benchmarking Llama-3.2 1B Model

Using our local testbed (§5.1), we evaluate OptiReduce with the Llama-3.2 1B model [65] on three well-known downstream tasks: SQuAD (extractive question answering) [129], ARC (science reasoning) [58], and MATH (symbolic mathematics) [83], across both low-tail ($P_{99/50} = 1.5$) and high-tail ($P_{99/50} = 3.0$) environments. Table 2 provides a detailed comparison of training times across all schemes. OptiReduce consistently demonstrates performance improvements across all tasks. Compared to NCCL, it achieves speedups of 1.35× on MATH, 1.25× on ARC, and 1.08× on SQuAD, averaging a 1.24× improvement. The gains are even more pronounced against Gloo, with speedups of 1.73×, 1.61×, and 1.49× respectively, averaging 1.61×. These improvements scale further under high-tail conditions, reaching speedups of up to 2.1× while preserving baseline model convergence and test accuracies.

C Network and Compute Intensive Models & Base LMs

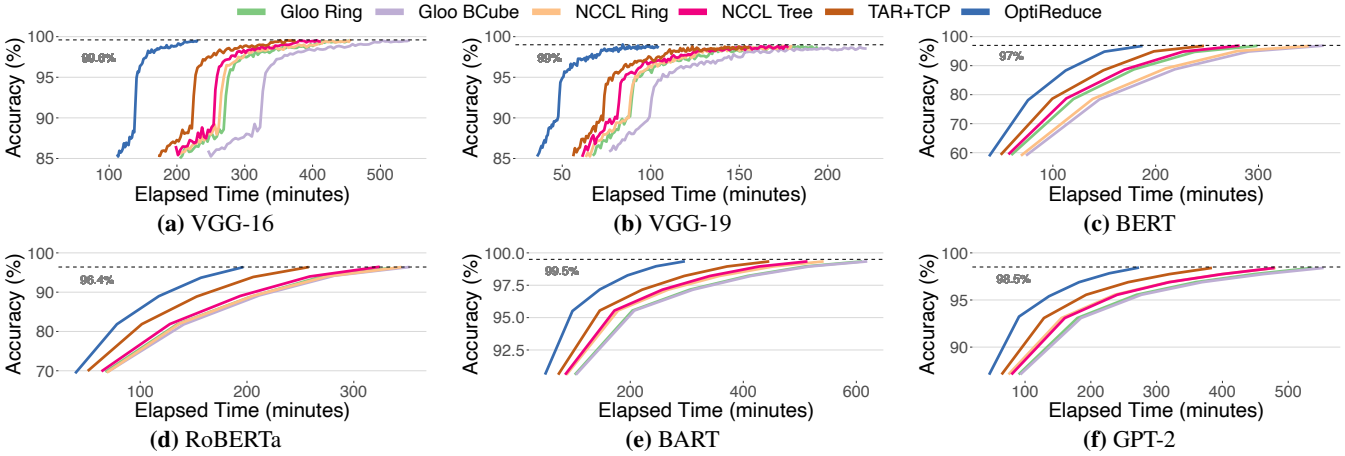
In this section, we present time-to-accuracy (TTA) plots for additional models, including computer vision models (ResNet-50/101/152, VGG-16/19) and base LMs (BERT, RoBERTa, BART, and GPT-2). The experiments use the same local testbed setup described in §5.1, but with six worker nodes (VMs). We compare results across two environment configurations, characterized by tail-to-median ratios ($P_{99/50}$) of 1.5



(a) VGG-16 (b) VGG-19 (c) BERT

(d) RoBERTa (e) BART (f) GPT-2

Figure 18: Time-to-accuracy (TTA) of baseline systems vs OptiReduce with tail-to-median ratio: $P_{99/50} = 1.5$.



(a) VGG-16 (b) VGG-19 (c) BERT

(d) RoBERTa (e) BART (f) GPT-2

Figure 19: Time-to-accuracy (TTA) of baseline systems vs OptiReduce with tail-to-median ratio: $P_{99/50} = 3$.

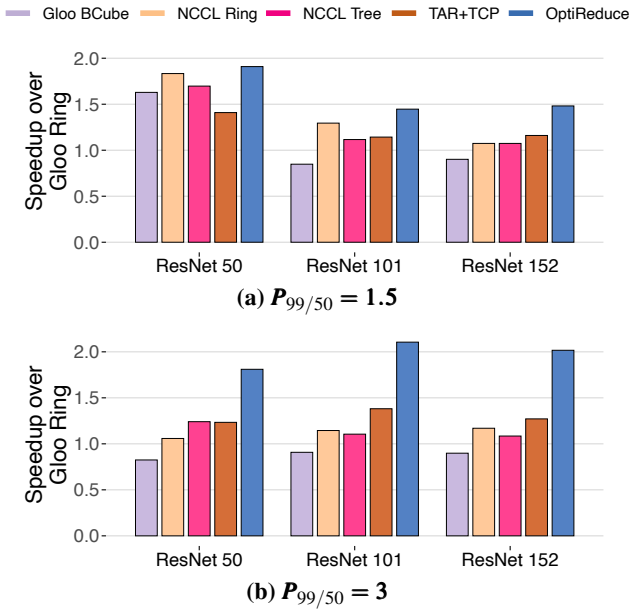
(low variability) and 3 (high variability).

C.1 Time-to-accuracy (TTA)

We observe similar gains for these network-intensive models (VGG-16/19) and base LMs, with up to (66%, 75%) and (50%, 51%) reductions in TTA, on average, compared to Gloo (Ring, BCube) and NCCL (Ring, Tree), respectively—Figure 18 ($P_{99/50} = 1.5$) and Figure 19 ($P_{99/50} = 3$). OptiReduce achieves the same convergence accuracy as the baselines while limiting lost gradients to less than 1.5%, on average, of the communicated traffic.

C.2 Training Throughput (Speedup)

While compute-intensive models like ResNets [82] typically do not gain significant advantages from optimized communication [103, 153], their performance can be impacted in shared environments (such as public clouds) due to long-tail latencies. Our evaluations reflect this, where OptiReduce demonstrates notable improvements over baseline systems, achieving average speedups of 22% over NCCL and 53% over Gloo for three ResNet models (50/101/152) across both environment configurations (Figure 20).



(a) $P_{99/50} = 1.5$

(b) $P_{99/50} = 3$

Figure 20: Training throughput for computationally-intensive ResNet models on the ImageNet dataset.