Straggler-Aware Scheduling for Distributed DNN Training

Joseph Ye
justdoye@mit.edu

Mahdi Afshari
afshari@mit.edu

6.5820 Networks for Machine Learning

Code: https://github.mit.edu/afshari/Straggler-Aware-Congestion-Control

Abstract

Distributed deep learning training requires frequent gradient synchronization across workers, where iteration time is bounded by the slowest flow. We present a straggler-aware scheduling approach that detects persistently lagging workers and dynamically reallocates communication rates to accelerate their recovery. Unlike traditional congestion control protocols that optimize for per-flow fairness, we optimize for collective completion time. A key challenge is distinguishing persistent stragglers from transient slowdowns; naive detection causes harmful oscillation under bursty conditions. We address this with persistence filtering, cooldown periods, and gradual recovery. Our prototype achieves ≈45% reduction in iteration time under persistent stragglers and <1% overhead under transient conditions.

1 Introduction

Modern deep neural networks require distributed training across multiple accelerators. Data-parallel training replicates the model across workers, each processing different batches before synchronizing gradients through collective operations. This synchronization creates a fundamental bottleneck: every worker must wait for the slowest one before proceeding.

Formally, if N workers participate in a synchronization barrier, iteration time equals:

$$T_{\text{iter}} = \max_{i \in [1,N]} T_{\text{flow}_i} \tag{1}$$

This max-of-flows property means a single slow worker dominates iteration time. We call such workers *stragglers*: those that consistently exceed a threshold above median completion time. Network congestion, particularly incast during gradient synchronization, creates stragglers through packet drops and retransmissions; we model the effect as extra delay on affected flows.

Existing datacenter congestion control protocols optimize for per-flow metrics. DCTCP [1] and TIMELY [2] treat flows equally, but per-flow fairness is actually harmful in this setting: a small slowdown in a single flow hurts the whole job, while speeding up already-fast flows does nothing. MLTCP [4] coordinates flows within ML jobs but still aims for balanced progress rather than explicit straggler acceleration.

A naive approach—detecting slow workers and immediately reallocating bandwidth—fails under transient conditions. If a worker experiences a brief slowdown (e.g., a microburst), reacting instantly penalizes other workers; by the next iteration the burst is gone but the penalty persists, causing oscillation.

We address this with persistence-aware scheduling: only reallocate when a worker is slow for multiple consecutive iterations, apply cooldowns between reallocations, and gradually restore allocations when workers recover. We optimize *per-iteration latency for a single job*, not cluster-wide metrics; multi-job fairness is left to future work. Our contributions:

- Persistence-filtered straggler detection that distinguishes transient from chronic slowdowns
- Adaptive reallocation with cooldown and gradual recovery to prevent oscillation
- Evaluation showing 45% improvement under persistent stragglers with <1% overhead under transient conditions

2 Background

2.1 Distributed Training Communication

In data-parallel training, worker i computes gradients g_i on its local batch, then all workers synchronize to obtain $\bar{g} = \frac{1}{N}\sum_{i=1}^{N} g_i$. Two architectures dominate:

Parameter Server: Workers send gradients to a central aggregator, creating many-to-one incast during the push phase.

All-Reduce: Workers collectively aggregate using ring all-reduce, exchanging with neighbors in $N-1$ stages. Even without pure incast, flows share paths and create queue buildup.

Both require barrier synchronization: the collective completes only when all participants finish.

2.2 The Incast Problem

Incast occurs when multiple senders transmit to a single receiver simultaneously. Workers finish computation at similar times, triggering synchronized transmission. Switch buffers overflow, dropping packets that require retransmission. Which flows experience drops is essentially random, so stragglers emerge unpredictably.

2.3 Limitations of Existing Protocols

DCTCP uses ECN marking to signal congestion while maintaining per-flow fairness. HPCC [3] leverages in-network telemetry for faster reaction. MLTCP recognizes that ML flows belong to the same job and coordinates their windows to achieve balanced progress across the job's own flows, but does not minimize job-level makespan. None aim to minimize $\max_i T_{\text{flow}_i}$ (Equation 1).

3 Design

3.1 Key Insight

Let workers have completion times $T_1 \leq T_2 \leq \cdots \leq T_N$. Reducing any T_i with $i < N$ leaves $\max_i T_i$ unchanged—those workers already wait. Only reducing T_N improves iteration time.

However, naive instant reallocation causes oscillation under transient conditions: (1) worker k is briefly slow due to a burst, (2) we penalize other workers, (3) burst ends but penalties persist, (4) penalized workers become new stragglers, (5) repeat.

3.2 Persistence-Filtered Detection

We only trigger reallocation when a worker is slow for K consecutive iterations:

$$\text{streak}_i^{(t)} = \begin{cases} \text{streak}_i^{(t-1)} + 1 & \text{if } T_i^{(t)} > (1+\theta) \cdot T_{\text{med}}^{(t)} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

$$\text{confirmed\_straggler}_i = \mathbf{1}\left[\text{streak}_i \geq K\right] \tag{3}$$

Note that T_{med} is recomputed every iteration, so the notion of "slow" is relative and adapts to background conditions.

Parameter choices. We use $K = 3$ and $\theta = 0.2$. The threshold $\theta = 0.2$ roughly matches where iteration time visibly degrades in our profiles (workers more than 20% slower than median consistently bottleneck the collective). We ablate K in Section 5.3; values below 3 cause oscillation under bursty conditions, while values above 5 delay detection unnecessarily under persistent stragglers.

Algorithm 1 Persistence-Aware Straggler Mitigation

Require: Workers W, threshold θ, persistence K, cooldown C

1: Initialize: delay$[i] \leftarrow d_{\text{base}}$, streak$[i] \leftarrow 0$, cd $\leftarrow 0$
2: **for** each training iteration **do**
3: Run all-reduce; record completion time T_i for each worker
4: $T_{\text{med}} \leftarrow \text{median}(\{T_i\})$
5: **for** $i \in W$ **do**
6: **if** $T_i > (1+\theta) \cdot T_{\text{med}}$ **then**
7: streak$[i] \leftarrow$ streak$[i] + 1$
8: **else**
9: streak$[i] \leftarrow 0$
10: delay$[i] \leftarrow \gamma \cdot$ delay$[i] + (1-\gamma) \cdot d_{\text{base}}$ ▷ Gradual recovery
11: **end if**
12: **end for**
13: $S \leftarrow \{i : \text{streak}[i] \geq K\}$
14: **if** $|S| > 0$ **and** $|S| < N$ **and** cd $= 0$ **then**
15: **for** $i \in W$ **do**
16: **if** $i \in S$ **then**
17: delay$[i] \leftarrow$ delay$[i]/(1+\alpha)$
18: **else**
19: delay$[i] \leftarrow$ delay$[i] \cdot (1 + \beta \cdot \frac{|S|}{N-|S|})$
20: **end if**
21: **end for**
22: cd $\leftarrow C$
23: **end if**
24: cd $\leftarrow \max(0, \text{cd} - 1)$
25: **end for**

3.3 Rate Adjustment with Cooldown

When stragglers are confirmed, we reallocate rates:

$$r_i' = \begin{cases} r_i(1+\alpha) & \text{if confirmed straggler} \\ r_i\left(1 - \beta \cdot \frac{|S|}{N-|S|}\right) & \text{otherwise} \end{cases} \tag{4}$$

We use $\alpha = 0.3$ (straggler speedup) and $\beta = 0.15$ (donor penalty). The asymmetry ($\alpha > 2\beta$) reflects a deliberate choice: helping stragglers aggressively is high-value since it directly reduces T_{iter}, while penalizing donors gently avoids creating new stragglers. Cooldown $C = 5$ iterations prevents rapid oscillation. All parameters represent a single global knob per worker, applied uniformly across all tensors in the collective.

3.4 Gradual Recovery

When a worker's streak resets, we gradually restore its allocation via exponential moving average: $r_i^{(t+1)} = \gamma \cdot r_i^{(t)} + (1-\gamma) \cdot r_{\text{base}}$ with $\gamma = 0.5$. Recovery is slower than punishment to prevent bounce-back oscillation. This update appears as line 10 in Algorithm 1.

4 Implementation

We implement our prototype in PyTorch using Gloo backend (∼1,800 lines of Python).

Custom All-Reduce: Ring all-reduce via point-to-point send/recv, enabling per-worker timing hooks.

Network Model: We model network heterogeneity as per-message delays via blocking sleeps. This is a significant simplification: there is no real incast, queue buildup, packet drops, or retransmissions. Our "rate" control manipulates delays directly ($d_i = d_{\text{base}}/r_i$) rather than interacting with a transport protocol. The goal is to test the control logic (persistence filtering, cooldown, recovery) rather than produce production-ready congestion control. Four delay profiles:

- **Uniform**: All workers sleep d_{base} (10ms)
- **Straggler**: One worker sleeps $3 \times d_{\text{base}}$ persistently
- **Variable**: Delays sampled from $\mathcal{N}(d_{\text{base}}, \sigma^2)$ per iteration
- **Bursty**: Random workers get 5× delay with 20% probability

Workload: Small CNN (4 conv layers, 2 FC layers, 201K parameters) on CIFAR-10, 4 workers. Communication accounts for ∼70% of iteration time. We verified that without synthetic delays, iteration time variance is <5ms, much smaller than our injected delays.

5 Evaluation

5.1 Methodology

Experiments run on a single machine using Python multiprocessing. Each configuration runs 5 times with different seeds (250 iterations each); we report mean and standard deviation across 1,240 iterations per configuration, excluding 2 warmup iterations per run. Statistical significance tested via Welch's t-test.

5.2 Main Results

All configurations reach comparable final accuracy; we focus on iteration time. Figure 1 shows our approach achieves **44.8% improvement** under the straggler profile (1298ms → 717ms, $p < 0.0001$). For variable profile, iteration times are 268ms (baseline) vs 268ms (ours); for bursty profile, 747ms vs 750ms. Both show <1% difference, confirming persistence filtering prevents harmful oscillation. All experiments use $N = 4$ workers; we leave exploration of larger N to future work.

Figure 2 shows the distribution shift under the straggler profile: nearly all iterations complete faster. The median improves by 46% and p99 by 27%.

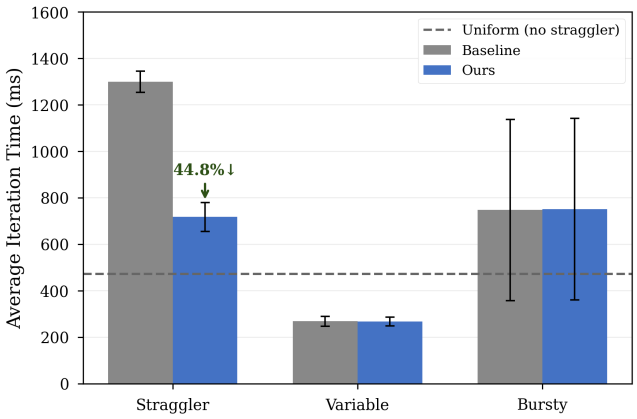

Figure 1: Average iteration time across delay profiles. Error bars show standard deviation across 5 seeds. Our approach achieves 44.8% improvement under persistent stragglers while maintaining <1% overhead under transient conditions.

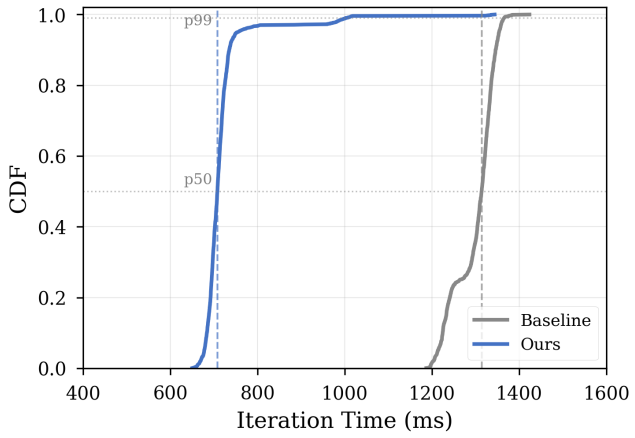

Figure 2: CDF of iteration times under straggler profile. Our approach shifts the entire distribution left, improving median (708ms vs 1314ms) and tail latency (p99: 1001ms vs 1365ms).

5.3 Ablation: Persistence Threshold

Figure 3 demonstrates why persistence filtering is critical. With $K = 1$ (instant reaction), the bursty profile shows **23% regression** (747ms → 878ms) due to 179 unnecessary reallocations causing oscillation. With $K = 3$, only 5 reallocations occur and performance matches baseline. On persistent stragglers, higher K mainly delays detection by a few iterations but does not hurt final performance.

6 Limitations

Simulated Network: Delays use `sleep()`, not actual congestion. No queue buildup, loss, or retransmissions.

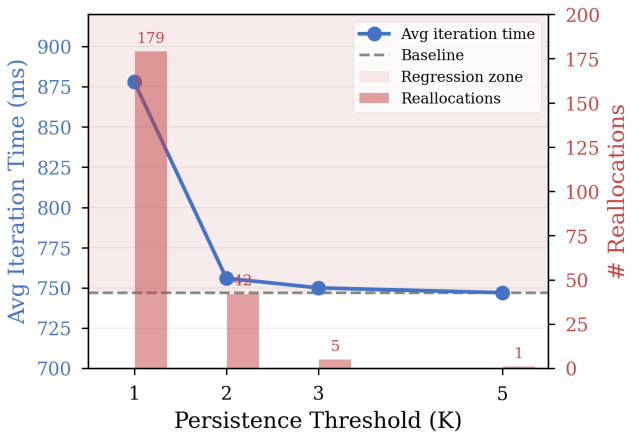

Figure 3: Effect of persistence threshold K on bursty profile. Shaded region indicates regression relative to baseline. $K = 1$ causes 23% regression from oscillation (179 reallocations); $K \geq 3$ achieves stability with minimal reallocations.

The incast dynamics motivating the problem are not actually exercised.

Single Machine: Workers run as processes, missing real datacenter network effects.

Single Job: We optimize iteration latency for one job; multi-job fairness is not addressed. Accelerating one job's straggler may hurt others sharing links.

Relative Detection: If all workers slow down together (e.g., background congestion), T_{med} shifts up and no one appears as a straggler. Our scheme only handles *relative* heterogeneity.

Compute vs Network: We do not distinguish compute-induced slowdowns from network-induced ones. If a worker is slow due to compute, our communication-side intervention is the wrong lever.

Iteration-Level Detection: Finer-grained within-collective monitoring could enable faster reaction while still filtering transient slowdowns.

Future work includes transport-layer implementation (using ECN or RTT signals to modulate cwnd or pacing rate), evaluation on real distributed clusters, and integration with multi-job schedulers.

7 Related Work

Datacenter CC: DCTCP [1] pioneered ECN-based congestion control. TIMELY [2] uses RTT gradients; HPCC [3] leverages INT. All optimize per-flow metrics.

ML-Aware Networking: MLTCP [4] coordinates ML job flows. SwitchML [5] performs in-network aggregation, requiring dataplane changes; our scheme operates purely at endpoints. ByteScheduler [6] optimizes tensor scheduling. Our persistence-filtered straggler detection could be layered on top of these systems: straggler scores could serve as priorities for ByteScheduler's tensor ordering or inform MLTCP's rate allocation.

Straggler Mitigation: MapReduce [7] handles compute stragglers via speculation. We address network-induced stragglers with rate reallocation.

8 Conclusion

We presented persistence-aware straggler scheduling for distributed training. By requiring consecutive slow iterations before reallocating, applying cooldowns, and gradually recovering, we achieve 45% improvement under persistent stragglers while avoiding oscillation under transient conditions. The key insight: distinguishing persistent from transient slowdowns is essential for stable, effective straggler mitigation.

References

[1] M. Alizadeh et al. Data Center TCP (DCTCP). *SIGCOMM*, 2010.

[2] R. Mittal et al. TIMELY: RTT-based Congestion Control. *SIGCOMM*, 2015.

[3] Y. Li et al. HPCC: High Precision Congestion Control. *SIGCOMM*, 2019.

[4] H. Xu et al. MLTCP: Congestion Control for DNN Training. 2024.

[5] A. Sapio et al. Scaling Distributed ML with In-Network Aggregation. *NSDI*, 2021.

[6] Y. Peng et al. Generic Communication Scheduler for DNN Training. *SOSP*, 2019.

[7] J. Dean, S. Ghemawat. MapReduce: Simplified Data Processing. *OSDI*, 2004.