BANDWIDTH-OPTIMAL ALLREDUCE UNDER NETWORK FAILURES

Anonymous authors
Paper under double-blind review

ABSTRACT

Network failures are among the most frequent hardware faults in large-scale GPU clusters and a leading cause of training-job interruptions. Modern collective communication libraries such as NCCL mitigate network failures by rerouting traffic through surviving NICs on the same server, trading reduced inter-node bandwidth for uninterrupted training. However, the degraded server remains on the critical path of the standard ring algorithm, slowing the *entire* collective. We derive information-theoretic lower bounds on AllReduce completion time under asymmetric network bandwidth and show that when the straggler retains at least half of its original bandwidth, the unavoidable overhead relative to the fault-free optimum is only $O(1/p)$ for p GPUs. We then design OPTCC, a four-stage pipelined AllReduce algorithm that approaches this lower bound. Experiments on SimAI confirm that OPTCC closes the gap left by existing fault-tolerant schemes: under practical network failures with up to 50% bandwidth loss and a single straggler, OPTCC completes AllReduce within 2–6% of NCCL's fault-free ring performance, whereas the state-of-the-art incurs up to 57% overhead.

1 INTRODUCTION

Network failures, including NIC, cable, and port faults, are among the most frequent hardware faults in large-scale GPU clusters, wasting 10–15% of GPU hours in production training runs Kokolis et al. (2025); Llama Team, AI @ Meta (2024); Gunawi et al. (2018); Diskin et al. (2021). Recent systems mitigate these failures by rerouting the traffic on these GPUs through surviving NICs on the same server via PXN NVIDIA Corporation (2022), keeping the training job alive without a full restart Lin et al. (2025); Si et al. (2025); Ryabinin et al. (2021) (Figure 2). However, this reduces the affected server's aggregate inter-node bandwidth, turning its GPUs into *stragglers*—GPUs whose effective network bandwidth is lower than their healthy peers. On mainstream multi-NIC servers such as those with H100 or B100 GPUs, losing a single NIC cuts bandwidth by 12.5%.

The collective most affected by this bandwidth degradation is the AllReduce used for data-parallel gradient synchronization Gibiansky (2017); Narayanan et al. (2021); Rajbhandari et al. (2020). Ring AllReduce Patarasuk & Yuan (2009); Thakur et al. (2005); NVIDIA (2024) is bandwidth-optimal on homogeneous links, but this optimality breaks down on heterogeneous topologies Zhao et al. (2026); a network failure creates precisely such asymmetry. After failover, existing systems resume the standard Ring AllReduce without adapting to the asymmetric bandwidth, causing congestion at the straggler's degraded link that throttles the entire collective. Even the most sophisticated optimization among these Wang et al. (2025a) still incurs up to 57% increase in AllReduce runtime when 50% of the bandwidth is lost on a single straggler. *Existing approaches leave a significant performance gap between fault-free AllReduce and the degraded setting; in this paper, we aim to close it.*

Our key insight is that a straggler's irreducible contribution to AllReduce is inherently *small*: to guarantee correctness, the straggler must transmit its own private data at least once, but all remaining data transfers can be handled entirely by healthy GPUs. Crucially, these two parts use physically separate links—the straggler's slow link and the healthy GPUs' fast links—so they can proceed in parallel. Since the straggler's bandwidth is only one out of p links in the cluster, a well-designed pipeline can absorb this loss with an overhead of only $O(1/p)$ relative to the fault-free optimum. This insight extends to the more common production scenario of multiple stragglers, as long as the

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

total bandwidth lost across all degraded nodes remains sufficiently smaller than the healthy GPUs' aggregate link bandwidth.

In summary, we make three contributions:

1. **Information-theoretic lower bound.** We prove that the unavoidable overhead of bandwidth degradation is surprisingly small (Table 1): when the straggler retains at least half of its original bandwidth, the overhead over the fault-free optimum is only $O(1/p)$—less than 1% at p=128 GPUs. We extend the bound to more general multiple-straggler and multi-GPU-per-server scenarios (Section 3).
2. **Algorithm approaching the lower bound.** We design OptCC, a four-stage pipelined AllReduce that approaches this lower bound with zero idle bubbles in the steady-state pipeline body. The schedule is a closed-form construction requiring no optimization solver, supporting online generation when a failure is detected (Section 4).
3. **SimAI-based evaluation.** On the SimAI[1] network simulator Wang et al. (2025b), OptCC completes AllReduce within 2–6% of the fault-free NCCL ring under practical network failures with a single straggler, whereas existing approaches incur up to 57% overhead (Section 5). Code is available at `https://anonymous.4open.science/r/OPTCC_algorithms-38B0/README.md`.

Table 1: Theoretical lower bounds on straggler overhead relative to the fault-free baseline T_0; p: number of GPUs; ℓ (resp. ℓ_i): straggler slowdown factor; $\ell_1 = \max_i \ell_i$. For one GPU per server, $T_0 = 2(p-1)n/p$; for g GPUs per server, $T_0 = 2(p-g)n/(gp)$.

	$\ell\ (\ell_1) < 2-\frac{1}{p-1}$	$\ell\ (\ell_1) \geq 2-\frac{1}{p-1}$
No straggler	T_0	T_0
Single straggler, 1 GPU/server in DP group (Thm. 1)	$\left(1+\frac{\ell-1}{\ell\, p}+O\left(\frac{1}{p^2}\right)\right)T_0$	$\left(\frac{\ell}{2}+\frac{\ell}{2\,p}+O\left(\frac{1}{p^2}\right)\right)T_0$
m **stragglers, 1 GPU/server in DP group** (Thm. 2)	$\left(1+\frac{\sum_i \frac{\ell_i-1}{\ell_i}}{p}+O\left(\frac{1}{p^2}\right)\right)T_0$	$\left(\frac{\ell_1}{2}+\frac{\ell_1}{2\,p}+O\left(\frac{1}{p^2}\right)\right)T_0$
Single straggler, g **GPUs/server in DP group** (Thm. 3)	$\left(1+\frac{g(\ell-1)}{\ell\, p}+O\left(\frac{g^2}{p^2}\right)\right)T_0$	$\left(\frac{\ell}{2}+\frac{\ell\, g}{2\,p}+O\left(\frac{g^2}{p^2}\right)\right)T_0$

Notes. (1) For m stragglers, the column header shows the bound expression only; the transition point depends on $\ell_1, \ldots, \ell_m$ (Appendix B.2). (2) Tighter bounds exist for the single-straggler and multi-GPU/server cases (Appendix B.1, Thm. 6; Appendix B.3, Thm. 13); see Appendix A for a full summary.

2 Background and Motivation

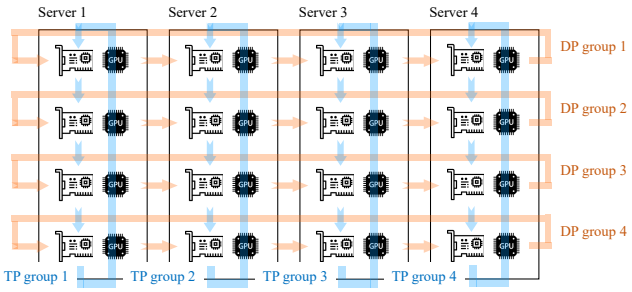


Figure 1: 2D parallelism communication topology example for large-model training (4 servers × 4 GPUs). Each data-parallel (DP) group consists of one GPU per server; gradient synchronization is performed via AllReduce over the inter-node RDMA fabric (IB/RoCE). Each tensor-parallel (TP) group consists of the GPUs within a single server, executing collectives over intra-node NVLink.

AllReduce topology under hybrid parallelism. Modern LLM training combines multiple parallelism strategies Narayanan et al. (2021); Huang et al. (2019); Narayanan et al. (2019); Zheng et al.

[1]SimAI is a large-scale, production-level network simulator that models GPU clusters and communication libraries to evaluate distributed training and collective communication under realistic network conditions.

(2022)—including tensor parallelism (TP), data parallelism (DP), and pipeline parallelism—of which TP and DP require AllReduce (Figure 1). Because TP demands an AllReduce at every layer during both the forward and backward passes, it is confined to GPUs within a single server to exploit the high-bandwidth NVLink fabric; DP AllReduce, which synchronizes gradients once per iteration, runs across servers over the inter-node RDMA network (IB/RoCE). Each DP group typically occupies only one (or a small number of) GPU and NIC per server, making DP AllReduce the collective directly exposed to inter-node network failures.

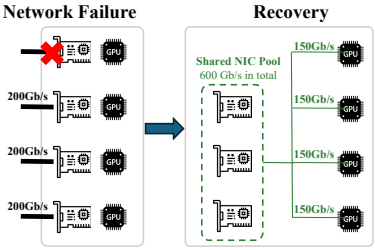


Figure 2: Network failure recovery: remaining NICs on this server form a *shared NIC pool*, turning the affected server into a *straggler* (e.g., losing 2 of 8 NICs gives slowdown factor $\ell = 4/3 \approx 1.33$).

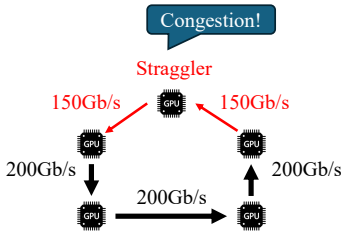


Figure 3: The straggler lies on the critical path of standard Ring AllReduce, bringing congestion to the entire collective.

Network failure recovery creates stragglers. When a network failure (e.g., NIC, cable, or port fault) occurs during training, there are two strategies to resume without interrupting training. The first is to *exclude* the failed server and continue with the remaining healthy servers Si et al. (2025); Zhao et al. (2024); Devraj et al. (2025); Tandon et al. (2017); this preserves homogeneous bandwidth but wastes the excluded server's GPUs. The second is to *keep* the failed server by rerouting its traffic through surviving network links (i.e., NICs) on the same server Wang et al. (2025a); Zhang et al. (2025); Lin et al. (2025). As illustrated in Figure 2, the PXN (PCI $\times$ NVLink) mechanism in NCCL NVIDIA Corporation (2022) allows any GPU to access any NIC on the same server via NVLink, effectively pooling all local NICs into a shared bandwidth resource. This keeps all GPUs active, but leaves the GPUs on the affected server as stragglers with degraded network bandwidth.

Ring AllReduce bottleneck under asymmetric bandwidth. When network bandwidth is the bottleneck—as is the case for large gradient buffers in data-parallel training[2]—NCCL selects the Ring AllReduce algorithm Patarasuk & Yuan (2009); NVIDIA (2024). Ring AllReduce arranges all GPUs in a logical ring and performs a reduce-scatter followed by an allgather, with each GPU sending and receiving approximately twice the message size in total. This is information-theoretically optimal when all links have equal bandwidth Patarasuk & Yuan (2009). However, when one server operates at reduced bandwidth, requiring every GPU—including the straggler—to send the same volume of data creates congestion at the straggler's degraded link (Figure 3). The straggler becomes the bottleneck of the entire ring, and all healthy GPUs are throttled to its pace. Existing liveness-oriented systems do not address this: after failover, they resume the standard Ring AllReduce without reducing the straggler's data obligation or restructuring the schedule. R^{2}CCL Wang et al. (2025a) takes a step further by adapting the schedule to the degraded topology, but a substantial performance gap to the fault-free baseline remains (Section 5). General-purpose schedule synthesis tools Cai et al. (2021); Shah et al. (2023); Zhao et al. (2026); Shapley et al. (2026) can in principle handle heterogeneous topologies, but their solver-based approach requires minutes to hours of computation even at moderate scale, making them unsuitable for online re-scheduling when a network failure occurs during training.

Problem settings of AllReduce topology. The topology of the DP AllReduce depends on the parallelism configuration and the number of concurrent failures; we study three settings that cover the practical spectrum:

- **Single straggler, one GPU per server in DP group.** In large-scale training of frontier models, all GPUs within a server are assigned to TP (and possibly PP), so each server contributes exactly

[2]We focus on the bandwidth term because (i) for the large gradient buffers in LLM training (hundreds of MB to several GB Gibiansky (2017)), bandwidth dominates AllReduce cost—NCCL itself selects the bandwidth-optimal Ring algorithm for sufficiently large messages NVIDIA (2024)—and (ii) per-hop network latency varies widely across GPU pairs in a data center and is difficult to model analytically Qian et al. (2024).

one GPU to each DP AllReduce group. A single network failure degrades one server, creating one straggler in the AllReduce ring.

- **Multiple stragglers, one GPU per server in DP group.** In large clusters with hundreds of servers, multiple servers may experience network failures and fall back to degraded networking Kokolis et al. (2025). This is the most common production scenario; it extends the above setting to handle heterogeneous slowdown factors across multiple degraded servers.
- **Single straggler, multiple GPUs per server in DP group.** For medium-scale training or fine-tuning, TP may use only a subset of GPUs within a server (e.g., 2 or 4), leaving multiple GPUs per server in the same DP group. These GPUs share the server's inter-node NIC bandwidth, so a network failure degrades the effective bandwidth for all of them simultaneously.

3 Theoretical Lower Bounds

We establish information-theoretic lower bounds on AllReduce time under degraded-NIC conditions, following the general methodology of communication lower bounds in distributed learning Huang et al. (2022). The results stated below correspond to the lower-bound entries in Table 1; all proofs are deferred to Appendix B.

Problem setting. Let $T(\mathcal{A})$ denote the end-to-end completion time of an AllReduce algorithm $\mathcal{A}$ on $p \geq 3$ GPUs with n-element vectors, counted in the bandwidth-bound model: each healthy NIC transmits one element per time unit, a NIC with slowdown $\ell > 1$ transmits one element in ℓ time units, and per-message latency, kernel launch, and cold-start overheads are excluded Patarasuk & Yuan (2009). An algorithm is *correct* if (i) every transfer respects data dependencies—a GPU may transmit only data it already holds—and (ii) upon termination every GPU holds the element-wise sum $\mathbf{s} = \sum_{i=1}^{p} \mathbf{x}_i$. An algorithm is *bandwidth-optimal* if it achieves $\min T$ over all correct AllReduce algorithms on the same bandwidth profile. In the fault-free homogeneous case with g GPUs per server, this minimum is known to equal[3]

$$T_0 \;=\; \frac{2(p-g)}{gp}\, n\,,$$

which reduces to $2n(p-1)/p$ when $g = 1$, and is attained by Ring AllReduce Patarasuk & Yuan (2009). We measure the overhead of all algorithms under degraded bandwidth relative to this T_0.

One straggler. Exactly one GPU has slowdown factor ℓ, and the remaining $p-1$ are healthy.

Theorem 1 (Lower bound with one straggler)**.** *Any correct AllReduce algorithm $\mathcal{A}$ satisfies*

$$T(\mathcal{A}) \;\geq\; \max\left\{ \frac{2\ell(p-1)}{\ell(p-1)+1},\; \ell \right\} \cdot n \;=\; \max\left\{ 1+\frac{\ell-1}{\ell\, p}+O\left(\frac{1}{p^2}\right),\; \frac{\ell}{2}+\frac{\ell}{2p}+O\left(\frac{1}{p^2}\right) \right\} \cdot T_0.$$

Multiple stragglers. More generally, m GPUs are stragglers with heterogeneous slowdown factors $\ell_1 \geq \ell_2 \geq \cdots \geq \ell_m > 1$; the remaining $p-m$ are healthy (p is sufficiently larger than m).

Theorem 2 (Lower bound with m stragglers)**.** *Any correct AllReduce algorithm $\mathcal{A}$ satisfies*

$$T(\mathcal{A}) \;\geq\; \max\left\{ \frac{2(p-1)}{p-m+\sum_i \frac{1}{\ell_i}},\; \ell_1 \right\} \cdot n \;=\; \max\left\{ 1+\frac{\sum_i \frac{\ell_i-1}{\ell_i}}{p}+O\left(\frac{1}{p^2}\right),\; \frac{\ell_1}{2}+\frac{\ell_1}{2p}+O\left(\frac{1}{p^2}\right) \right\} \cdot T_0.$$

Setting $m = 1$ recovers Theorem 1.

Multiple GPUs per server. Now we consider each server holding g GPUs that share the same NIC bandwidth. Inside each server, the g GPUs are connected via NVLink whose bandwidth is at least $(g-1)$ times the NIC bandwidth.[4] The straggler server's inter-server NIC is degraded by factor ℓ, while all NVLink connections remain healthy.

[3] When $g > 1$, this is attained by NCCL's hierarchical scheme NVIDIA (2024): an inter-server ring AllReduce over the p/g servers on n/g elements per rank, bracketed by intra-server NVLink reduce-scatter and all-gather.

[4] On the NVIDIA DGX A100 NVIDIA Corporation (2020), each GPU has 4800 Gbps aggregate NVLink bandwidth (2400 Gbps unidirectional) versus 200 Gbps inter-node NIC bandwidth (one HDR InfiniBand NIC per GPU), so this assumption holds by a wide margin.

Theorem 3 (Lower bound with multi-GPU servers)**.** *Under the NVLink-rich assumption above, any correct AllReduce algorithm $\mathcal{A}$ satisfies*

$$T(\mathcal{A}) \;\geq\; \max\left\{ \frac{2\ell(p-g)}{g\,(\ell(p-g)+g)},\;\; \frac{\ell}{g} \right\}\cdot n \;=\; \max\left\{ 1+\frac{g(\ell-1)}{\ell\,p}+O\!\left(\frac{g^2}{p^2}\right),\;\; \frac{\ell}{2}+\frac{\ell\,g}{2\,p}+O\!\left(\frac{g^2}{p^2}\right) \right\}\cdot T_0\,.$$

Setting $g = 1$ recovers Theorem 1.

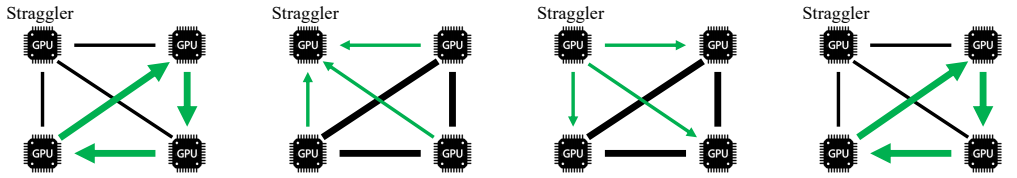
Takeaway. In all three settings, the lower bound has the form $T \geq \max\{1 + O(1/p),\ \ell/2 + O(1/p)\} \cdot T_0$. When the worst straggler loses more than half of its bandwidth, it bottlenecks the entire AllReduce and any algorithm requires at least $(\ell/2)\,T_0$ time. When no straggler loses more than half of its bandwidth ($\ell < 2$), the lower bound approaches T_0 as p grows, meaning the overhead from NIC degradation vanishes in large clusters.

4 Algorithm Design

Our algorithm partitions each GPU's n-element vector into many small segments and, for every segment, decomposes the AllReduce into four logical stages: (1) reduce-scatter among the $p-1$ healthy GPUs, (2) upload partial sums to the straggler, (3) download global sums from the straggler, and (4) allgather among the healthy GPUs. Stages 2 and 3 use only the slow straggler link while Stages 1 and 4 use fast links between healthy GPUs—these are *disjoint hardware resources*. We exploit this by pipelining segments so that the slow Stages 2/3 of one segment run in parallel with Stages 1/4 of other segments, effectively hiding the straggler overhead. We present the algorithm for the single-straggler, one-GPU-per-server setting in this section; the algorithm designs for multiple stragglers (Appendix D) and multi-GPU-per-server (Appendix E) are deferred to the appendix.

4.1 Four-Stage Decomposition

We partition the n input elements into k *segments*, each containing $p-1$ *sections* of size $s = n/(k(p-1))$ elements. For each segment, the AllReduce proceeds through four stages (illustrated in Figure 4). We define a *flow* as a point-to-point transfer of one section between two GPUs, and enforce that each NIC receives at most one flow at a time; overlapping incoming flows would contend for bandwidth and cause unpredictable delays.



Stage1: Reduce-scatter between healthy GPUs | Stage 2: Upload to the straggler | Stage 3: Download from the straggler | Stage 4: Allgather between healthy GPUs

Figure 4: 4-stage decomposition of OptCC. Line thickness indicates bandwidth; *arrows* denote *active* links in each stage.

Stage 1: Reduce-Scatter among healthy servers. The $p-1$ healthy servers are arranged in a directed ring. Each section is reduce-scattered along this ring in exactly $p-2$ hops: at each hop, a server receives a partial-sum flow from its predecessor, adds its own contribution, and forwards the result. After $p-2$ hops, one designated healthy server holds the partial sum of all $p-1$ healthy servers' contributions for that section.

Since all $p-1$ servers send in parallel (each to its ring successor), and each hop transmits one section of s elements on a fast link, the time for Stage 1 is $(p-2)\cdot s$.

Stages 2/3: Upload and download at the straggler. After Stage 1, the partial sum of a section resides on a healthy server but does not yet include the straggler's data. In Stage 2, a healthy server sends the partial sum to the straggler (s elements on the slow link, taking $\ell \cdot s$ time); in Stage 3, the straggler adds its local value, producing the global sum, and sends it back to a healthy server (s elements, again $\ell \cdot s$ time).

Each segment has $p-1$ sections, and the flow scheduling constraint forces the straggler to process them one at a time: no two flows may overlap at the straggler's NIC. The total time for Stages 2 and 3 for one segment is therefore $(p-1)\cdot\ell\cdot s$ each. Because the straggler can receive and send in full-duplex, Stage 2 and Stage 3 for *different* sections can be pipelined on the straggler's link.

Stage 4: Allgather among healthy servers. Once a section's global sum returns to a healthy server via Stage 3, it is broadcast to all other healthy servers through an allgather along the same ring. This is structurally symmetric to Stage 1: $p-2$ hops, each forwarding s elements, costing $(p-2)\cdot s$.

Stage ordering and correctness. For a single section, the data dependencies (S1 $\rightarrow$ S2 $\rightarrow$ S3 $\rightarrow$ S4) admit two orderings that both produce a correct AllReduce: (1) S1 $\rightarrow$ S2 $\rightarrow$ S3 $\rightarrow$ S4, which reduce-scatters on the healthy ring, uploads to the straggler, and downloads for allgather; and (2) S3 $\rightarrow$ S1 $\rightarrow$ S4 $\rightarrow$ S2, in which the straggler uploads first, the healthy servers reduce-scatter with it, allgather, and finally return the result to the straggler.

4.2 Schedule Construction

A key observation is that Stages 2 and 3 communicate exclusively over the slow straggler link, whereas Stages 1 and 4 communicate exclusively over the fast ring links among healthy servers. Although individual GPUs may serve as endpoints in both types of transfer, the *links* involved are disjoint. This separation of resources makes it possible to overlap Stages 2/3 with Stages 1/4: if we execute them concurrently, each parallel body takes time proportional to $\max\{\ell,\,2\}\cdot s$, which aligns precisely with the lower bound established in Theorem 1.

Four patterns. Exploiting the two valid stage orderings identified in Section 4.1, we design four *patterns*—distinct flow schedules that can be interleaved without violating the non-overlap constraint. We illustrate the construction for $p=5$ and $\ell=2$ (Figure 5); the analysis generalizes to any p and ℓ.

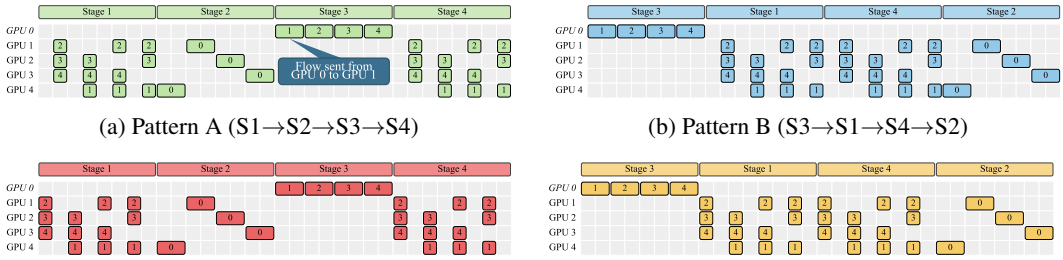
In this setting, GPU 0 is the straggler with 50% NIC throughput; GPUs 1–4 are healthy. Each Stage 2 or Stage 3 flow occupies $\ell\,s=2s$ time units (two grid cells in the figure), while each Stage 1 or Stage 4 flow occupies s time units (one grid cell). With $p-1=4$ healthy servers, Stage 1 (reduce-scatter among 4 GPUs) and Stage 4 (allgather among 4 GPUs) each consist of $(p-2)(p-1)=12$ flows; Stage 2 and Stage 3 each consist of $p-1=4$ flows. We reserve $2(p-1)=8$ time slots per stage (one grid cell equals the time for a healthy NIC to send or receive one flow; a straggler flow occupies two cells), for a total of $4\times 2(p-1)=32$ time slots per pattern across all four stages. The four patterns are:

- **Pattern A** (Figure 5a): S1 $\rightarrow$ S2 $\rightarrow$ S3 $\rightarrow$ S4.
- **Pattern B** (Figure 5b): S3 $\rightarrow$ S1 $\rightarrow$ S4 $\rightarrow$ S2.
- **Pattern C** (Figure 5c): S1 $\rightarrow$ S2 $\rightarrow$ S3 $\rightarrow$ S4 (same stage ordering as A, but flows in Stages 1 and 3 are shifted by an offset).
- **Pattern D** (Figure 5d): S3 $\rightarrow$ S1 $\rightarrow$ S4 $\rightarrow$ S2 (same ordering as B, with the same offset).

Each pattern respects the data dependencies of its stage ordering, and the four patterns are staggered so that all four stages interleave across time—no NIC ever receives or sends two simultaneous flows.

Pipelining four patterns. We now overlay the four patterns to form a complete schedule (Figure 6). We assume k is a multiple of 4, so that each pattern processes exactly $k/4$ segments. Superimposing the four patterns' flow schedules yields a composite timeline in which each parallel body contains one instance of each stage from each pattern.

Filling straggler-slot bubbles ($\ell<2$). When $\ell<2$, each Stage 2/3 flow ($\ell\,s$) is shorter than the combined Stage 1/4 flows ($2s$), leaving a bubble of $(2-\ell)\,s$ time in every straggler slot. Across one parallel body the straggler has $p-1$ such bubbles, and each healthy GPU also has one (in its Stage 2/3 slot). We reclaim these idle intervals by running a *point-to-point allreduce* between the healthy GPUs and the straggler (Figure 7): in parallel body b, each healthy GPU uses its bubble to send a partial sum to the straggler (light gray cells); the straggler reduces these contributions during its own bubbles and, in body $b+1$, broadcasts the result back to the healthy GPUs (dark gray cells). Because both directions reuse time that would otherwise be idle, a fraction of the data completes its allreduce at zero additional cost, bringing the effective runtime closer to the lower bound.



(a) Pattern A (S1→S2→S3→S4)

(b) Pattern B (S3→S1→S4→S2)

(c) Pattern C (S1→S2→S3→S4)

(d) Pattern D (S3→S1→S4→S2)

Figure 5: Flow schedules for the four patterns (p=5, ℓ=2; *GPU 0* is the straggler with 50% bandwidth). Rows are GPUs; columns are time slots; each cell is a flow labeled by its destination. The four patterns use disjoint slots so no NIC receives two flows at once.

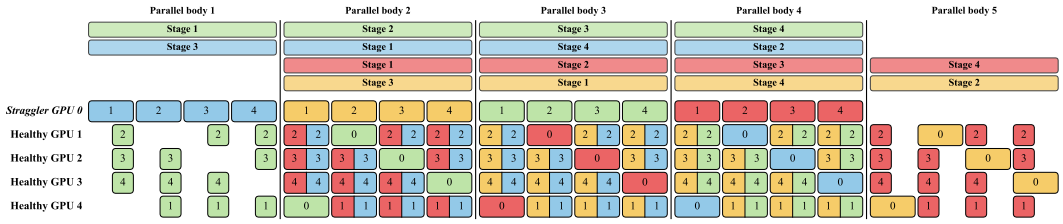


Figure 6: **Combining the four patterns of Figure 5 into a complete pipeline** (p=5, ℓ=2, k=4). Each parallel body (except head/tail) contains all four stages. Flows in a rounded box share the same sender–receiver pair with no dependency and are packed into one transfer. **When ℓ=2 the body has no idle bubbles.**

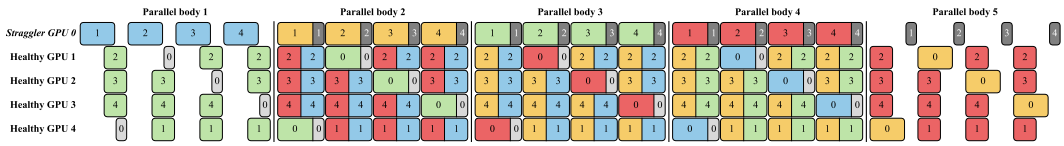


Figure 7: **Filling all bubbles with a P2P allreduce between healthy GPUs and the straggler when** ℓ<2 (p=5, ℓ=1.5, k=4). Stage 2/3 flows (width 1.5) are shorter than Stage 1/4 flows (width 2), creating bubbles. light-gray cells (body i): healthy GPUs send partial sums to the straggler; dark-gray (body i+1): straggler broadcasts reduced results back.

4.3 Time Analysis

Let $s = n/(k(p-1))$ denote the section size. The composite schedule is formed by concatenating $k/4$ copies of the four-pattern group from Figure 6 (for $\ell \geq 2$) or Figure 7 (for $\ell < 2$) end-to-end, where the last parallel body of one group overlaps with the first of the next. This produces $k+1$ parallel bodies in total: $k-1$ steady-state bodies that each contain one part from each of the four patterns, plus one startup body at the head and one drain body at the tail. The body duration depends on which resource—the healthy ring (Stages 1/4) or the straggler link (Stages 2/3)—is the bottleneck.

Case $\ell \geq 2$ (straggler-bottlenecked). When $\ell \geq 2$, Stages 2/3 are the critical path. Each parallel body takes $\ell\,(p-1)\,s$ time. Summing over $k+1$ bodies (including startup/drain), substituting $s = n/(k(p-1))$, and recalling the fault-free baseline $T_0 = 2n(p-1)/p$:

$$T \;=\; \ell\, n\, \frac{k+1}{k} \;\xrightarrow{k\to\infty}\; \ell\, n \;=\; \left(\frac{\ell}{2} + \frac{\ell}{2\,p} + O\!\left(\frac{1}{p^2}\right)\right) T_0\,, \tag{1}$$

matching the lower bound of Theorem 1 (Table 1). When $\ell = 2$, Stages 2/3 fit exactly within Stages 1/4 and every slot is occupied with zero idle time. When $\ell > 2$, the straggler becomes the bottleneck, yet the *schedule remains bandwidth-optimal without requiring* $p \to \infty$, confirming its optimality even for small clusters.

Case $\ell < 2$ (ring-bottlenecked, with bubble filling). When $\ell < 2$, the bubble-filling technique of Section 4.2 reduces the section size from s to a smaller s', and uses up all the straggler's bandwidth. The derivation of s' and the resulting total time are given in Appendix C:

$$T = \frac{2(p-1)\,\ell}{(p-2)\ell+2}\, n\, \frac{k+\ell-1}{k} \xrightarrow{k\to\infty} \frac{2(p-1)\,\ell\, n}{(p-2)\ell+2} = \left(1 + \frac{2(\ell-1)}{\ell\, p} + O\left(\frac{1}{p^2}\right)\right) T_0\,. \quad (2)$$

At $\ell = 1$ this reduces to T_0, the non-degraded optimum; at $\ell = 2$ it gives $2n$, approaching the lower bound of Theorem 1. For general $\ell \in [1, 2)$, the relative overhead satisfies $p\ell/[(p-2)\ell+2] \le 1 + 1/(p-1)$. In fact, our algorithm always *approaches bandwidth optimal without requiring* $p \to \infty$; see Appendix C for details.

Schedule generation complexity. The complete flow schedule is determined by the closed-form rules of Sections 4.1–4.2 in $O(pk)$ time, with no iterative optimization. For p=1024 GPUs on SimAI, the schedule is generated in under 1 ms on a single CPU core, making OptCC suitable for online deployment upon failure detection.

4.4 Extensions to Multiple Stragglers and Multi-GPU Servers

Multiple stragglers, one GPU per server in DP group (Appendix D). The key difference is that Stages 2/3 now serve m stragglers instead of one; the bottleneck is the slowest straggler ($\ell_1 = \max_i \ell_i$), and flows to/from each straggler are assigned to disjoint time slots within each body.

Single straggler, multiple GPUs per server in DP group (Appendix E). We treat each server's g GPUs as a single *logical GPU*: before every outgoing NIC flow the g GPUs perform a local reduce over NVLink, and after every incoming flow the receiver broadcasts to the other $g-1$ GPUs. Because NVLink bandwidth is typically $(g-1)\times$ the NIC bandwidth, these intra-server transfers can be fully *overlapped* with inter-server NIC transfers of adjacent pipeline stages, so the NIC remains the sole bottleneck and intra-server communication adds no extra time to the critical path.

5 Evaluation

We evaluate OptCC on SimAI Wang et al. (2025b) (an NS-3-based network simulator that models the actual on-the-wire traffic between GPUs), using clusters of A100 servers, each equipped with 8 GPUs and 8 ConnectX-6 200 Gbps HDR InfiniBand NICs.[5] We compare against three baselines: **$\text{NCCL}_{\text{NoFailure}}$**, NCCL ring on a fully healthy topology (a fault-free reference); **ICCL** Zhang et al. (2025), NCCL ring on the NIC-bandwidth-degraded topology; and **R^2CCL** Wang et al. (2025a), a state-of-the-art NIC fault-tolerant AllReduce. We also plot the **lower bounds** from Section 3 as theoretical runtime guarantees.[6] In the OptCC implementation, flow dependencies are enforced via per-GPU state tracking for intra-GPU ordering and necessary P2P synchronization messages for inter-GPU ordering. We simulate at production-realistic GPU counts Llama Team, AI @ Meta (2024); Narayanan et al. (2021) and AllReduce message sizes N, sweeping N from 128 MiB to 2 GiB Jiang et al. (2024); Fernandez et al. (2025). SimAI is deterministic—the same configuration always produces the same result—so each data point is a single run with no error bars.

OptCC runtime is close to $\text{NCCL}_{\text{NoFailure}}$ in both small and large clusters. OptCC stays within 6% of $\text{NCCL}_{\text{NoFailure}}$ across $p \in [16, 256]$ under a single straggler. At small p (e.g., p=16) the gap is dominated by the straggler-induced lower bound of Theorem 1 — fundamental, not an artifact of OptCC. Toward p=256 we observe a mild upward trend; we honestly acknowledge this as a limitation of our construction, since larger clusters require more point-to-point synchronization between healthy and straggler GPUs, whose overhead grows with p. Even so, at p=256 OptCC stays within 6% of $\text{NCCL}_{\text{NoFailure}}$ and beats R^2CCL/ICCL by 29%/57%. Note that ICCL's overhead exceeds the slowdown factor ℓ itself: because ICCL does not restructure the ring schedule, the straggler's degraded NIC must carry the same volume of traffic as a healthy NIC, causing network congestion at the straggler that amplifies the bandwidth penalty beyond the raw $\ell\times$ slowdown.

[5] We disabled NVIDIA SHARP to ensure our results generalize to diverse hardware configurations.

[6] SimAI profiling shows that the NIC bandwidth term accounts for ∼86% of $\text{NCCL}_{\text{NoFailure}}$'s completion time, with the remaining ∼14% from fixed overhead (cold start, per-flow network latency). We plot the lower bound as its multiplier applied to the bandwidth fraction, plus the fixed overhead.

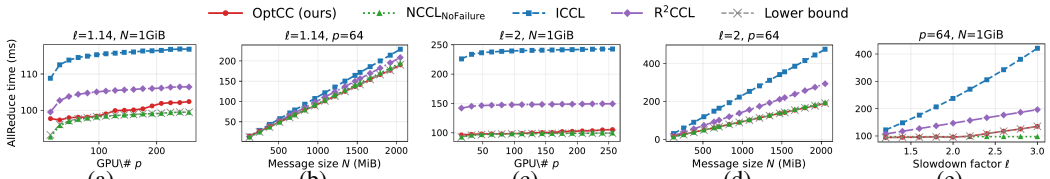


Figure 8: Single straggler; 1 GPU per server in DP group. (a,b) straggler loses 1/8 NICs (ℓ=1.14); (c,d) loses 4/8 NICs (ℓ=2); (e) varying ℓ.

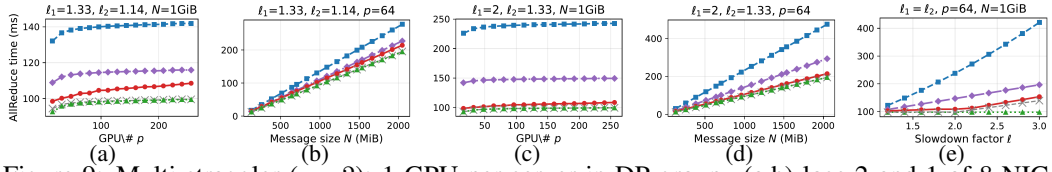


Figure 9: Multi-straggler (m=2); 1 GPU per server in DP group. (a,b) lose 2 and 1 of 8 NICs (ℓ_1=1.33, ℓ_2=1.14); (c,d) lose 4 and 2 of 8 NICs (ℓ_1=2, ℓ_2=1.33); (e) ℓ_1=ℓ_2, varying ℓ.

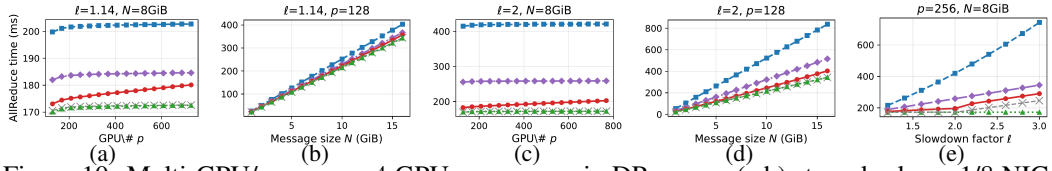


Figure 10: Multi-GPU/server; g=4 GPUs per server in DP group. (a,b) straggler loses 1/8 NICs (ℓ=1.14); (c,d) loses 4/8 NICs (ℓ=2); (e) varying ℓ.

OptCC approaches NCCL$_{\text{NoFailure}}$ across all AllReduce message sizes. As long as the worst NIC retains at least 50% of its bandwidth, OptCC matches the NCCL$_{\text{NoFailure}}$ rate to within 2% (single straggler), 8.5% (multi-straggler m=2), and 4% (multi-GPU/server), in each case beating R^{2}CCL and ICCL at the same N. The constant-factor gap translates linearly into wall-clock savings as gradient buffers grow.

OptCC approaches NCCL$_{\text{NoFailure}}$ whenever the worst NIC loses at most 50% of its bandwidth. For $\ell \leq 2$ the OptCC curve sits flat against NCCL$_{\text{NoFailure}}$: bubble filling (Section 4.2) hides the entire straggler overhead inside slack on the healthy ring. Once ℓ crosses 2 the straggler's NIC becomes the binding bottleneck and OptCC grows linearly at Theorem 1's $\ell\, n$ rate; even so, OptCC beats ICCL and R^{2}CCL, which both pay a slowdown penalty from ℓ=1 onward.

End-to-end training-iteration overhead. To connect the collective-level speedup to training throughput, we simulate one iteration of GPT-175B pretraining under ℓ=2 following the MegaScale-175B configuration Jiang et al. (2024) (8,192 A100 GPUs, TP=8/PP=8/DP=128). Relative to NCCL$_{\text{NoFailure}}$, OptCC caps the failure-induced iteration overhead at 0.45%, versus 4.88% for ICCL and 2.44% for R^{2}CCL—a $10.8\times$ and $5.4\times$ reduction in the iteration-time overhead, respectively. Full configuration and per-method throughput in Appendix G.

6 Conclusion

Network failures are an operational reality in large GPU clusters, yet their impact on collective communication has lacked a principled theoretical treatment. This paper closes that gap for AllReduce: we derive tight information-theoretic lower bounds showing that, when the straggler retains at least half of its bandwidth, the unavoidable overhead is only $O(1/p)$—small in large clusters—and we design OptCC, an algorithm that approaches these bounds. The key insight is that healthy GPUs carry enough spare bandwidth to absorb a straggler's deficit through careful pipeline interleaving, so that near-optimal performance is attainable *without any hardware redundancy, job restart, or offline pre-computation*. SimAI-based evaluation confirms 2–6% overhead relative to the fault-free NCCL ring with a single straggler, compared with up to 57% for existing schemes. Extending the bounds to latency-sensitive settings and dynamic bandwidth fluctuations remains future work.

AI USE STATEMENT

In this work, we used generative AI tools solely for editing the paper for readability and language polish (a recommended-disclosure task under the ICLR 2027 AI Policy). We did *not* use generative AI tools for any required-disclosure task: the theoretical results, mathematical claims, proof arguments, algorithm design, research methodology, experimental setup, implementation, and interpretation of results were all developed by the authors without AI assistance. We have reviewed all AI-assisted work; every AI-polished passage was read, verified, and edited by the authors, and we take responsibility for the final content of this work, including all text, claims, and artifacts.

ETHICS STATEMENT

This work studies bandwidth-optimal collective communication algorithms for large-scale distributed training. It does not involve human subjects, personally identifiable data, or datasets that raise privacy, fairness, or discrimination concerns. All experiments are carried out in the SimAI simulator rather than on shared production infrastructure, so no additional operational risks were introduced during evaluation. The proposed algorithm is general-purpose training infrastructure and, to the best of our knowledge, does not create foreseeable avenues of misuse beyond those already inherent to training large-scale models.

REPRODUCIBILITY STATEMENT

To support reproducibility, we release an anonymous open-source implementation of the OptCC algorithm together with the scripts required to reproduce the theoretical constructions and experimental sweeps reported in this paper:

`https://anonymous.4open.science/r/OPTCC_algorithms-38B0/README.md`

Complete proofs of the lower bounds appear in Appendix B. The algorithm construction and its time analysis are detailed in Section 4 and Appendix D. The experimental protocol (SimAI configuration, cluster topology, message sizes, and straggler patterns) is described in Section 5.

REFERENCES

Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. Synthesizing optimal collective algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 62–75, 2021.

Arjun Devraj, Eric Ding, Abhishek Vijaya Kumar, Robert Kleinberg, and Rachee Singh. Efficient AllReduce with stragglers. *arXiv preprint arXiv:2505.23523*, 2025.

Michael Diskin, Alexey Bukhtiyarov, Max Ryabinin, Lucile Saulnier, Quentin Lhoest, Anton Sinitsin, Dmitry Popov, Dmitry V. Pyrkin, Maxim Kashirin, Alexander Borzunov, Albert Villanova del Moral, Denis Mazur, Ilia Kobelev, Yacine Jernite, Thomas Wolf, and Gennady Pekhimenko. Distributed deep learning in open collaborations. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (eds.), *Advances in Neural Information Processing Systems*, volume 34, pp. 7879–7897. Curran Associates, Inc., 2021. URL `https://proceedings.neurips.cc/paper_files/paper/2021/file/41a60377ba920919939d83326ebee5a1-Paper.pdf`.

Jared Fernandez, Luca Wehrstedt, Leonid Shamis, Mostafa Elhoushi, Kalyan Saladi, Yonatan Bisk, Emma Strubell, and Jacob Kahn. Efficient hardware scaling and diminishing returns in large-scale training of language models. *Transactions on Machine Learning Research*, 2025.

Andrew Gibiansky. Bringing HPC techniques to deep learning. `https://andrew.gibiansky.com/blog/machine-learning/baidu-allreduce/`, 2017.

Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Deepthi Srinivasan, Biswaranjan Panda, Andrew Baptist, Gary Grider, Parks M. Fields, Kevin Harms, Robert B.

Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. *ACM Trans. Storage*, 14(3), October 2018. ISSN 1553-3077. doi: 10.1145/3242086. URL https://doi.org/10.1145/3242086.

Xinmeng Huang, Yiming Chen, Wotao Yin, and Kun Yuan. Lower bounds and nearly optimal algorithms in distributed learning with communication compression. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 18955–18969. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/77f2d0c271e508278ea13e24cd8773d5-Paper-Conference.pdf.

Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and Zhifeng Chen. GPipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. MegaScale: Scaling large language model training to more than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024.

Apostolos Kokolis, Michael Kuchnik, John Hoffman, Adithya Kumar, Parth Malani, Faye Ma, Zachary DeVito, Shubho Sengupta, Kalyan Saladi, and Carole-Jean Wu. Revisiting reliability in large-scale machine learning research clusters. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 1259–1274, 2025. doi: 10.1109/HPCA61900.2025.00096.

Shengkai Lin, Kairui Zhou, Hongtao Zhang, Yibo Wu, Yi Pan, Yihan Yang, Qinwei Yang, Wei Zhang, Arvind Krishnamurthy, and Shizhen Zhao. SHIFT: Exploring the boundary of RDMA network fault tolerance. *arXiv preprint arXiv:2512.11094*, 2025.

Llama Team, AI @ Meta. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. PipeDream: Generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on GPU clusters using Megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2021.

NVIDIA. NCCL tests. https://github.com/NVIDIA/nccl-tests, 2024.

NVIDIA Corporation. NVIDIA DGX A100 system architecture. https://images.nvidia.com/aem-dam/Solutions/Data-Center/nvidia-dgx-a100-datasheet.pdf, 2020. Accessed: 2026-05-03.

NVIDIA Corporation. Doubling all2all performance with NVIDIA collective communication library 2.12. https://developer.nvidia.com/blog/doubling-all2all-performance-with-nvidia-collective-communication-library-2-12/, 2022. Accessed: 2026-03-31.

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

Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, Chao Wang, Peng Wang, Pengcheng Zhang, Xianlong Zeng, Eddie Ruan, Zhiping Yao, Ennan Zhai, and Dennis Cai. Alibaba HPN: A data center network for large language model training. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, pp. 691–706, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706141. doi: 10.1145/3651890.3672265. URL https://doi.org/10.1145/3651890.3672265.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020. ISBN 9781728199986.

Max Ryabinin, Eduard Gorbunov, Vsevolod Plokhotnyuk, and Gennady Pekhimenko. Moshpit SGD: Communication-efficient decentralized training on heterogeneous unreliable devices. In *Advances in Neural Information Processing Systems*, volume 34, 2021.

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 *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2023.

Richard Shapley, Rachit Agarwal, and David Shmoys. Optccl: Scalable synthesis of optimal collective communication algorithms. In *Proceedings of the ACM SIGCOMM 2026 Conference*, pp. 177–189, 2026.

Min Si, Pavan Balaji, Yongzhou Chen, Ching-Hsiang Chu, Adi Gangidi, Saif Hasan, Subodh Iyengar, Dan Johnson, Bingzhe Liu, Regina Ren, Deep Shah, Ashmitha Jeevaraj Shetty, Greg Steinbrecher, Yulun Wang, Bruce Wu, Xinfeng Xie, Jingyi Yang, Mingran Yang, Kenny Yu, Minlan Yu, Cen Zhao, Wes Bland, Denis Boyda, Suman Gumudavelli, Prashanth Kannan, Cristian Lumezanu, Rui Miao, Zhe Qu, Venkat Ramesh, Maxim Samoylov, Jan Seidel, Srikanth Sundaresan, Feng Tian, Qiye Tan, Shuqiang Zhang, Yimeng Zhao, Shengbao Zheng, Art Zhu, and Hongyi Zeng. Collective communication for 100k+ GPUs. *arXiv preprint arXiv:2510.20171*, 2025.

Rashish Tandon, Qi Lei, Alexandros G. Dimakis, and Nikos Karampatziakis. Gradient coding: Avoiding stragglers in distributed learning. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*, pp. 3368–3376, 2017.

Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in MPICH. *International Journal of High Performance Computing Applications*, 19(1): 49–66, 2005.

Wei Wang, Nengneng Yu, Sixian Xiong, and Zaoxing Liu. Reliable and resilient collective communication library for LLM training and serving. *arXiv preprint arXiv:2512.25059*, 2025a.

Xizheng Wang, Qingxu Li, Yichi Xu, Gang Lu, Dan Li, Li Chen, Heyang Zhou, Linkang Zheng, Sen Zhang, Yikai Zhu, Yang Liu, Pengcheng Zhang, Kun Qian, Kunling He, Jiaqi Gao, Ennan Zhai, Dennis Cai, and Binzhang Fu. SimAI: Unifying architecture design and performance tuning for Large-Scale large language model training with scalability and precision. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pp. 541–558, Philadelphia, PA, April 2025b. USENIX Association. ISBN 978-1-939133-46-5. URL https://www.usenix.org/conference/nsdi25/presentation/wang-xizheng-simai.

Mingjun Zhang, Xiaohe Hu, Menghao Zhang, Ziteng Chen, Yanmin Jia, Yan Zhang, Da Liu, Qing Chen, Fangzheng Jiao, Jun Chen, et al. An efficient, reliable and observable collective communication library in large-scale gpu training clusters. *arXiv preprint arXiv:2510.00991*, 2025.

Liangyu Zhao, Saeed Maleki, Yuanhong Wang, Zezhou Wang, Ziyue Yang, Hossein Pourreza, and Arvind Krishnamurthy. ForestColl: Throughput-Optimal collective communications on heterogeneous network fabrics. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, pp. 2067–2093, 2026.

Xiaoyang Zhao, Zhe Zhang, and Chuan Wu. AdapCC: Making collective communication in distributed machine learning adaptive. In *Proceedings of the 44th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2024. doi: 10.1109/ICDCS60910.2024.00012.

Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E Gonzalez, and Ion Stoica. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

A Summary of Optimality Results

This appendix collects the theoretical lower bounds, algorithm time complexities, and optimality results presented throughout the paper. Table 2 summarizes the lower bounds, Table 3 summarizes the algorithm time complexities, and Table 4 summarizes the optimality results.

Recall the problem setting. Let p be the total number of GPUs, each holding an n-element vector. A healthy NIC transmits one element per time unit, while a NIC with slowdown factor $\ell > 1$ requires ℓ time units per element. Upon termination, every GPU holds the element-wise sum $\mathbf{s} = \sum_{i=1}^{p} \mathbf{x}_i$. Let g be the number of GPUs per server—interconnected via NVLink with $g-1$ times the bandwidth of a single NIC—and let m be the number of straggler servers. By the PXN mechanism (Section 2), all NICs on a given server share the same slowdown factor. We consider three scenarios:

1. single straggler, one GPU per server (m=1, g=1);
2. multiple stragglers, one GPU per server (fixed m, g=1), with slowdown factors $\ell_1 \geq \ell_2 \geq \cdots \geq \ell_m > 1$;
3. single straggler, multiple GPUs per server (fixed g, m=1), where the total number of servers is $q = p/g$.

A.1 Best Lower Bounds

The best known lower bounds for these three scenarios are given by Theorems 6, 2, and 13, respectively; with full proofs deferred to Appendix B. We summarize these results in Table 2.

Table 2: Best known lower bounds on the AllReduce time T for three straggler settings. Each bound takes the maximum of two branches; the column headers indicate the dominant branch. Here $y_0 := 2(p-1)\big/\big(p - m + \sum_{i=1}^{m} 1/\ell_i\big)$.

Setting	**Small-ℓ regime**	**Large-ℓ regime**
Single straggler **One GPU per server** Theorem 6	$(1 < \ell < 2)$ $T \;\geq\; \dfrac{2\ell(p-1)}{\ell(p-2)+2} \cdot n$	$(\ell \geq 2)$ $T \;\geq\; \ell \cdot n$
Multiple stragglers **One GPU per server** Theorem 2	$(1 \leq \ell_1 < y_0)$ $T \;\geq\; \dfrac{2(p-1)}{p - m + \sum_{i=1}^{m} \frac{1}{\ell_i}} \cdot n$	$(\ell_1 \geq y_0)$ $T \;\geq\; \ell_1 \cdot n$
Single straggler **Multiple GPUs per server** Theorem 13	$(1 < \ell < 2)$ $T \;\geq\; \dfrac{2\ell(q-1)}{\ell(q-2)+2} \cdot \dfrac{n}{g}$	$(\ell \geq 2)$ $T \;\geq\; \ell \cdot \dfrac{n}{g}$

Notes. (1) For the single-straggler cases (rows 1 and 3), the two branches coincide at $\ell = 2$. (2) For m stragglers, the transition point $\ell_1 = y_0$ depends jointly on all $\ell_1 \geq \ell_2 \geq \cdots \geq \ell_m$ and p. (3) The bounds in rows 1 and 3 are *tight*: matching algorithms exist for all p (resp. q); see Table 3.

A.2 Algorithm Time Complexities

The algorithms for the three scenarios are presented in Section 4, Appendix D, and Appendix E, respectively. Their time analyses appear in Section 4.3 (together with Appendix C), Appendix D.3, and Appendix E.4, respectively. We summarize these in Table 3.

Table 3: Achieved AllReduce time T (as $k \to \infty$) for the three algorithm variants. Each column shows the dominant regime; the transition occurs at $\ell = 2$ in all cases.

Setting	**Small-ℓ regime**	**Large-ℓ regime**
Single straggler **One GPU per server** Section 4.3, Appendix C	$(1 < \ell < 2)$ $T \to \frac{2(p-1)\,\ell}{(p-2)\ell+2} \cdot n$	$(\ell \geq 2)$ $T \to \ell \cdot n$
Multiple stragglers **One GPU per server** Appendix D.3	$(1 < \ell_1 < 2)$ $T \to \frac{2(p-1)}{p-m} \cdot n$	$(\ell_1 \geq 2)$ $T \to \left(\ell_1 + \frac{2(m-1)}{p-m}\right) \cdot n$
Single straggler **Multiple GPUs per server** Appendix E.4	$(1 < \ell < 2)$ $T \to 2 \cdot \frac{n}{g}$ $\left(\text{or } \frac{2\ell(q-1)}{\ell(q-2)+2} \cdot \frac{n}{g}\right)^*$	$(\ell \geq 2)$ $T \to \ell \cdot \frac{n}{g}$

* The second time complexity is achieved via the bubble-filling technique described in Appendix E.4. Although theoretically possible, bubble filling is not implemented in practice.

A.3 Optimality Results

Combining the lower bounds from Table 2 with the algorithm times from Table 3 yields the optimality results summarized in Table 4. Here, **optimal** means that as $k \to \infty$ with p, m, g, and all ℓ_i held constant, the algorithm achieves bandwidth optimality, confirming its optimality even for small clusters. **Near-optimal** means that as $k \to \infty$ and $p \to \infty$ with m, g, and all ℓ_i held constant, the algorithm approaches bandwidth optimality, confirming near-optimality in large clusters.

Table 4: Optimality of the three algorithm variants (as $k \to \infty$).

Setting	**Small-ℓ regime**	**Large-ℓ regime**
Single straggler **One GPU per server**	$(1 < \ell < 2)$ Optimal	$(\ell \geq 2)$ Optimal
Multiple stragglers **One GPU per server**	$(1 < \ell_1 < 2)$ Near-optimal†	$(\ell_1 \geq 2)$ Near-optimal†
Single straggler **Multiple GPUs per server**	$(1 < \ell < 2)$ Near-optimal (or Optimal)*	$(\ell \geq 2)$ Optimal

† For Row 2, we expect that both the current algorithm and the lower bound in the small-ℓ regime of Table 2 can be further improved, and the branching point may change accordingly. The lower bound in the large-ℓ regime is expected to be tight for sufficiently large ℓ_1.

* The optimality result is achieved via the bubble-filling technique described in Appendix E.4. Although theoretically possible, bubble filling is not implemented in practice.

B Proof of Information-theoretic Lower Bounds

In this appendix, we prove all the theoretical lower bounds introduced in Section 3: Theorem 1 (Appendix B.1), Theorem 2 (Appendix B.2), and Theorem 3 (Appendix B.3). We further derive tighter bounds for the single-straggler case and the multi-GPU server setting: Theorem 6 sharpens Theorem 1, and Theorem 13 sharpens Theorem 3.

B.1 Single Straggler Case

We first prove the bounds for the single-straggler case: Theorem 1 and its tightened version, Theorem 6.

Consider $p \geq 3$ GPUs performing an AllReduce on vectors of n elements. Exactly one GPU—the straggler P_{s}—has a degraded NIC with slowdown factor $\ell > 1$: it transmits one element in ℓ time units. The remaining $p-1$ GPUs are healthy, each transmitting one element in one time unit. Let F denote the total number of element-transfers during an execution. Let $F_{\mathrm{s}}^{\mathrm{send}}$ and $F_{\mathrm{s}}^{\mathrm{recv}}$ denote the total number of elements sent and received by the straggler GPU P_{s}.

Theorem 1 (restated)**.** *Any correct AllReduce algorithm satisfies*

$$T \;\geq\; \max\left\{ \frac{2\ell(p-1)}{1+\ell(p-1)},\;\; \ell \right\} \cdot n\,.$$

Lemma 4 (Total traffic)**.** *Any correct AllReduce algorithm satisfies $F \geq 2(p-1)\,n$.*

Proof. Apply Lemma 10 to the special case where each GPU forms its own group, i.e., $q = p$. □

Lemma 5 (Straggler incident volume)**.** *Any correct AllReduce algorithm satisfies $F_{\mathrm{s}}^{\mathrm{send}} \geq n$ and $F_{\mathrm{s}}^{\mathrm{recv}} \geq n$.*

Proof. For each coordinate $j \in [n]$, $x_{P_{\mathrm{s}},j}$ is initially known only to P_{s} while every other GPU must obtain $s_j = \sum_i x_{i,j}$; hence information about coordinate j must leave P_{s} at least once. Also, P_{s} must output s_j, which depends on inputs initially outside P_{s}; hence information about coordinate j must enter P_{s} at least once. Summing over j gives the claim. □

Proof of Theorem 1. Let

$$F_{\mathrm{s}}^{\mathrm{send}} = x\,n, \qquad F_{\mathrm{s}}^{\mathrm{recv}} = y\,n,$$

with $x \geq 1$ and $y \geq 1$ by Lemma 5. Let $F_{\mathrm{h}\to\mathrm{h}}$ be the total number of element-transfers from healthy GPUs to healthy GPUs (counted as directed transfers), and define $F_{\mathrm{h}\to\mathrm{s}}$ and $F_{\mathrm{s}\to\mathrm{h}}$ analogously. Then

$$F = F_{\mathrm{h}\to\mathrm{h}} + F_{\mathrm{h}\to\mathrm{s}} + F_{\mathrm{s}\to\mathrm{h}}, \quad F_{\mathrm{h}\to\mathrm{s}} = F_{\mathrm{s}}^{\mathrm{recv}} = y\,n, \quad F_{\mathrm{s}\to\mathrm{h}} = F_{\mathrm{s}}^{\mathrm{send}} = x\,n.$$

Straggler bottlenecks. Every transfer incident to P_{s} costs ℓ, hence

$$T \geq \ell\, F_{\mathrm{s}}^{\mathrm{send}} = \ell x\, n. \tag{3}$$

Similarly,

$$T \geq \ell\, F_{\mathrm{s}}^{\mathrm{recv}} = \ell y\, n. \tag{4}$$

Healthy bottlenecks. Ignoring the extra time cost of healthy–straggler transfers on the healthy side, the total *sending* work across the $p-1$ healthy GPUs is at least $F_{\mathrm{h}\to\mathrm{h}} + F_{\mathrm{h}\to\mathrm{s}}$, so

$$T \geq \frac{F_{\mathrm{h}\to\mathrm{h}} + F_{\mathrm{h}\to\mathrm{s}}}{p-1} = \frac{F - F_{\mathrm{s}\to\mathrm{h}}}{p-1}.$$

Using $F \geq 2(p-1)n$ (Lemma 4) and substituting $F_{\mathrm{s}\to\mathrm{h}} = xn$ yields

$$\frac{T}{n} \geq 2 - \frac{x}{p-1}. \tag{5}$$

Similarly, the total *receiving* work across healthy GPUs is at least $F_{\mathrm{h}\to\mathrm{h}} + F_{\mathrm{s}\to\mathrm{h}}$, which implies

$$\frac{T}{n} \geq 2 - \frac{y}{p-1}. \tag{6}$$

Min–max form and symmetry reduction. Combining (3), (4), (5), and (6), for all $x \geq 1, y \geq 1$,

$$\frac{T}{n} \geq \max\left\{2 - \frac{x}{p-1},\ 2 - \frac{y}{p-1},\ \ell x,\ \ell y\right\}.$$

Let $g(x, y)$ denote the right-hand side. It is convex (a max of affine functions) and symmetric under swapping x and y, hence the minimum over $x \geq 1, y \geq 1$ is attained on the diagonal $x = y = z \geq 1$. Substituting $x = y = z$,

$$\frac{T}{n} \geq \min_{z \geq 1} \max\left\{2 - \frac{z}{p-1},\ \ell z\right\}.$$

Solving the 1-D minimization. The first branch is decreasing in z, while the second is increasing. Their intersection is given by

$$2 - \frac{z}{p-1} = \ell z,$$

namely

$$z^\star = \frac{2(p-1)}{1 + \ell(p-1)}.$$

If $z^\star \geq 1$, equivalently $\ell \leq 2 - \frac{1}{p-1}$, then the minimum is attained at $z^\star$ and

$$\frac{T}{n} \geq \ell z^\star = \frac{2\ell(p-1)}{1 + \ell(p-1)}.$$

If $z^\star < 1$, equivalently $\ell > 2 - \frac{1}{p-1}$, then over the feasible region $z \geq 1$ the minimum is attained at $z = 1$, which gives

$$\frac{T}{n} \geq \max\left\{2 - \frac{1}{p-1},\ \ell\right\} = \ell.$$

Therefore,

$$T \ \geq \ n \cdot \begin{cases} \dfrac{2\ell(p-1)}{1 + \ell(p-1)}, & 1 < \ell \leq 2 - \dfrac{1}{p-1}, \\ \ell, & \ell > 2 - \dfrac{1}{p-1}. \end{cases}$$

Equivalently,

$$T \ \geq \ \max\left\{\frac{2\ell(p-1)}{1 + \ell(p-1)},\ \ell\right\} \cdot n.$$

□

Note: Theorem 1 ignores the extra time cost of healthy–straggler transfers on the healthy side. Accounting for this cost yields a strictly stronger lower bound, which is in fact tight with respect to p; see Appendix C for the matching upper bound and discussion.

Theorem 6 (Tight Lower Bound: Single Straggler, One GPU per Server)**.** *Any correct AllReduce algorithm satisfies*

$$T \ \geq \ \max\left\{\frac{2\ell(p-1)}{\ell(p-2) + 2},\ \ell\right\} \cdot n\,.$$

Proof of Theorem 6. We follow the notation introduced in the proof of Theorem 1. Thus,

$$F_{\mathrm{s}}^{\mathrm{send}} = x\,n, \qquad F_{\mathrm{s}}^{\mathrm{recv}} = y\,n,$$

with $x \geq 1$ and $y \geq 1$, and

$$F = F_{\mathrm{h}\to\mathrm{h}} + F_{\mathrm{h}\to\mathrm{s}} + F_{\mathrm{s}\to\mathrm{h}}, \quad F_{\mathrm{h}\to\mathrm{s}} = y\,n, \quad F_{\mathrm{s}\to\mathrm{h}} = x\,n.$$

Straggler bottlenecks. Every transfer incident to P_{s} costs ℓ, hence

$$T \geq \ell\, F_{\mathrm{s}}^{\mathrm{send}} = \ell x\, n. \tag{7}$$

Similarly,

$$T \geq \ell\, F_{\mathrm{s}}^{\mathrm{recv}} = \ell y\, n. \tag{8}$$

Healthy bottlenecks (time-weighted). Healthy→healthy transfers cost 1 per element, while any transfer between a healthy GPU and the straggler costs ℓ per element. Therefore the total *sending* work across the $p-1$ healthy GPUs is at least $F_{\mathrm{h}\to\mathrm{h}} + \ell F_{\mathrm{h}\to\mathrm{s}}$, so

$$T \geq \frac{F_{\mathrm{h}\to\mathrm{h}} + \ell F_{\mathrm{h}\to\mathrm{s}}}{p-1} = \frac{F - F_{\mathrm{s}\to\mathrm{h}} + (\ell-1)F_{\mathrm{h}\to\mathrm{s}}}{p-1}.$$

Using $F \geq 2(p-1)n$ and substituting $F_{\mathrm{s}\to\mathrm{h}} = xn$, $F_{\mathrm{h}\to\mathrm{s}} = yn$ yields

$$\frac{T}{n} \geq 2 + \frac{(\ell-1)y - x}{p-1}. \tag{9}$$

Similarly, the total *receiving* work across healthy GPUs is at least $F_{\mathrm{h}\to\mathrm{h}} + \ell F_{\mathrm{s}\to\mathrm{h}}$, which implies

$$\frac{T}{n} \geq 2 + \frac{(\ell-1)x - y}{p-1}. \tag{10}$$

Min–max form and symmetry reduction. Combining (7), (8), (9), and (10), for all $x \geq 1, y \geq 1$,

$$\frac{T}{n} \geq \max\left\{2 + \frac{(\ell-1)y - x}{p-1},\ 2 + \frac{(\ell-1)x - y}{p-1},\ \ell x,\ \ell y\right\}.$$

Let $g(x,y)$ denote the right-hand side. It is convex (a max of affine functions) and symmetric under swapping x and y, hence the minimum over $x \geq 1, y \geq 1$ is attained on the diagonal $x = y = z \geq 1$. Substituting $x = y = z$,

$$\frac{T}{n} \geq \min_{z\geq 1} \max\left\{2 + \frac{(\ell-2)z}{p-1},\ \ell z\right\}.$$

Solving the 1-D minimization. If $\ell \geq 2$, both branches are nondecreasing in z, so the minimum is at $z = 1$ and $\frac{T}{n} \geq \ell$. If $1 < \ell \leq 2$, the minimum is attained at the intersection $2 + \frac{(\ell-2)z}{p-1} = \ell z$, i.e.,

$$z^\star = \frac{2(p-1)}{\ell(p-2)+2}, \qquad \frac{T}{n} \geq \ell z^\star = \frac{2\ell(p-1)}{\ell(p-2)+2}.$$

Therefore,

$$T \ \geq\ n \cdot \begin{cases} \ell, & \ell \geq 2, \\ \dfrac{2\ell(p-1)}{\ell(p-2)+2}, & 1 < \ell \leq 2. \end{cases}$$

Equivalently,

$$T \ \geq\ \max\left\{\frac{2\ell(p-1)}{\ell(p-2)+2},\ \ell\right\} \cdot n.$$

□

B.2 Multiple Stragglers Case

Here we prove the bound for the multiple-straggler case, Theorem 2.

Consider the same setting as above, but now m of the p GPUs are stragglers with heterogeneous slowdown factors $\ell_1 \geq \ell_2 \geq \cdots \geq \ell_m > 1$; the remaining $p-m$ are healthy. Let F denote the total number of element-transfers, and let $F_{\mathrm{s},i}^{\mathrm{send}}$ denote the total elements sent by straggler i.

Theorem 2 (restated)**.** *Any correct AllReduce algorithm satisfies*

$$T \ \geq\ \max\left\{\frac{2(p-1)}{p-m+\sum_{i=1}^{m}\frac{1}{\ell_i}},\ \ell_1\right\} \cdot n\,.$$

The proof relies on the following two lemmas.

Lemma 7 (Total traffic bound)**.** *Any correct AllReduce algorithm satisfies $F \geq 2(p-1)\,n$.*

Proof. Identical to Lemma 4. □

Lemma 8 (Straggler send volume bound)**.** *For each straggler i, any correct AllReduce algorithm satisfies $F_{\mathrm{s},i}^{\mathrm{send}} \geq n$.*

Proof. Identical to Lemma 5: each straggler's private data must leave it via at least n transfers. □

Proof of Theorem 2. Write $F_{\mathrm{s},i}^{\mathrm{send}} = x_i\, n$ with $x_i \geq 1$ (by Lemma 8). We derive $m+1$ bottleneck inequalities on T.

Straggler bottlenecks. Each straggler i sends $x_i\, n$ elements at rate $1/\ell_i$:

$$T \;\geq\; \ell_i\, x_i\, n \qquad \text{for each } i = 1, \ldots, m\,. \tag{11}$$

Healthy-GPU bottleneck. The $p-m$ healthy GPUs collectively send at least $F - \sum_i x_i\, n \geq 2(p-1)\, n - (\sum_i x_i)\, n$ elements (by Lemma 7). Averaging,

$$T \;\geq\; \frac{\left[2(p-1) - \sum_{i=1}^{m} x_i\right] n}{p-m}\,. \tag{12}$$

Combining. Taking the maximum of all $m+1$ bottlenecks,

$$T \;\geq\; g(x_1, \ldots, x_m) \;:=\; \max\left\{ \frac{\left[2(p-1) - \sum_i x_i\right] n}{p-m},\;\; \ell_1\, x_1\, n,\;\; \ldots,\;\; \ell_m\, x_m\, n \right\}.$$

This holds for every $(x_1, \ldots, x_m)$ with $x_i \geq 1$. Define

$$y_0 \;:=\; \frac{2(p-1)}{p - m + \sum_{i=1}^{m} \dfrac{1}{\ell_i}}$$

and $x_i^\star := y_0/\ell_i$. These are obtained by equating all $m+1$ terms of g: setting $\ell_i\, x_i = y$ for all i (giving $x_i = y/\ell_i$) and solving $[2(p-1) - y \sum_i 1/\ell_i]/(p-m) = y$ yields $y = y_0$.

The condition $x_i^\star \geq 1$ for all i reduces to $y_0 \geq \ell_1$ (since $\ell_1 \geq \ell_i$, if $x_1^\star = y_0/\ell_1 \geq 1$ then $x_i^\star \geq 1$ for all i).

Case 1 ($y_0 \geq \ell_1$, i.e., all $x_i^\star \geq 1$): The minimum of g on $\{x_i \geq 1\}$ is attained at $(x_1^\star, \ldots, x_m^\star)$ with value $y_0\, n$.

Case 2 ($y_0 < \ell_1$, i.e., $x_1^\star < 1$): The balanced solution is infeasible. Since g is a max of terms including $\ell_1\, x_1\, n \geq \ell_1\, n$, the minimum of g on $\{x_i \geq 1\}$ is at least $\ell_1\, n$.

Combining both cases,

$$T \;\geq\; \min_{x_i \geq 1} g(x_1, \ldots, x_m) \;=\; \max\left\{ \frac{2(p-1)}{p - m + \sum_{i=1}^{m} \frac{1}{\ell_i}},\;\; \ell_1 \right\} \cdot n\,,$$

completing the proof. □

Remark 9. *Unlike the other two subsections in this section, the refinement of the healthy-side bottleneck does not extend directly to the present setting with multiple bad servers. The reason is that, once the inter-server traffic is decomposed according to healthy and bad endpoints, one now obtains four directed classes,*

$$F_{\mathrm{h}\to\mathrm{h}}, \qquad F_{\mathrm{h}\to\mathrm{b}}, \qquad F_{\mathrm{b}\to\mathrm{h}}, \qquad F_{\mathrm{b}\to\mathrm{b}},$$

rather than three. In particular, the additional bad-to-bad traffic $F_{\mathrm{b}\to\mathrm{b}}$ is not captured, so the same strengthening trick used in the previous two subsections cannot be applied here in a direct way.

Consistency with the single-straggler bound. Setting $m = 1$ recovers $T \geq \max\{2\ell(p-1)/(\ell(p-1)+1),\; \ell\} \cdot n$, which coincides with Theorem 1.

B.3 MULTI-GPU PER SERVER CASE

Finally, we prove the bounds for the multi-GPU server setting with a single straggler: Theorem 3 and its tightened version, Theorem 13.

Write $q := p/g$ for the number of servers. We treat each server as a supernode. Inter-server communication uses NICs; intra-server communication uses NVLink. After normalization, each healthy NIC has unit rate, so each healthy server has aggregate NIC capacity g. One server S_s is slow: each of its NICs is degraded by factor ℓ, hence S_s has aggregate NIC capacity g/ℓ. Let F be the total traffic, F_{nic} the inter-server NIC traffic, $F_{\text{s}}^{\text{send}}$ the NIC traffic sent by S_s, and $F_{\text{nv}} = F - F_{\text{nic}}$ the NVLink traffic. (We do not need a lower bound on F_{nv}; ignoring NVLink can only weaken a lower bound on the runtime.)

NIC-time model. We count *directed* element-transfers. Any inter-server element-transfer that is incident to S_s takes ℓ time units; any inter-server transfer between healthy servers takes 1 time unit. Each server can send/receive at aggregate rate g (healthy) or g/ℓ (slow). Therefore, the total time T must be at least the corresponding (time-weighted) NIC workload divided by the relevant aggregate capacity.

Theorem 3 (restated)**.** *Any correct AllReduce algorithm satisfies*

$$T \;\geq\; \frac{n}{g} \cdot \max\left\{ \frac{2\ell(q-1)}{1+\ell(q-1)},\; \ell \right\}.$$

Lemma 10 (Inter-group traffic per coordinate via supernodes)**.** *Partition the p GPUs into $q \geq 1$ disjoint groups $\mathcal{G}_1, \ldots, \mathcal{G}_q$. Count only* inter-group element-transfers*: a transfer contributes 1 to coordinate j iff it carries the j-th coordinate from some GPU in $\mathcal{G}_u$ to some GPU in $\mathcal{G}_v$ with $u \neq v$.*

Then any correct AllReduce algorithm satisfies that for every coordinate $j \in [n]$, the number of inter-group element-transfers carrying coordinate j is at least $2(q-1)$. Consequently, the total inter-group traffic satisfies $F_{\text{inter}} \geq 2(q-1)n$.

Proof. Fix a coordinate $j \in [n]$ and let $s_j = \sum_{i=1}^{p} x_{i,j}$.

Supernode flow graph. For each time t, define a directed multigraph $H_{\leq t}^{(j)}$ on vertex set $[q] = \{1, \ldots, q\}$ (the groups as *supernodes*) as follows: for every inter-group element-transfer of coordinate j that completes by time t from a sender in group $\mathcal{G}_u$ to a receiver in group $\mathcal{G}_v$, add a directed edge $u \to v$.

For a group index $r \in [q]$ and time t, say that *group r can compute s_j at time t* if there exists some GPU in $\mathcal{G}_r$ whose local state at time t uniquely determines s_j (equivalently, there exists a decoding function of its local state that outputs s_j for all inputs).

Let

$$t_j \;=\; \inf\{t : \exists r \in [q] \text{ such that group } r \text{ can compute } s_j \text{ at time } t\},$$

and define the set of *roots at the first completion time*

$$A_j \;=\; \{r \in [q] : \text{group } r \text{ can compute } s_j \text{ at time } t_j\}, \qquad a_j = |A_j| \;(\geq 1).$$

Key reachability property. Fix $r \in A_j$. Then every group $u \in [q]$ must have a directed path $u \rightsquigarrow r$ in $H_{\leq t_j}^{(j)}$; otherwise the state within group r at time t_j is independent of the inputs from $\mathcal{G}_u$ (for coordinate j), so by an indistinguishability argument r cannot uniquely determine s_j at time t_j. In particular, for any chosen $r_0 \in A_j$, all u reach r_0. Choose an arbitrary $r_0 \in A_j$. Since all vertices reach r_0 in $H_{\leq t_j}^{(j)}$, the graph contains a spanning in-arborescence T_{in} rooted at r_0, hence

$$|E(T_{\text{in}})| \geq q - 1.$$

"Last-hop at t_j" into each root and disjointness. For every $r \in A_j$, there exists an inter-group edge e_r completing *exactly* at time t_j whose head is r; otherwise group r would receive no new inter-group information about coordinate j at time t_j, so r would already be able to compute s_j at some $t < t_j$, contradicting minimality of t_j. The edges $\{e_r\}_{r \in A_j}$ are pairwise distinct (different heads).

Any edge of T_{in} whose head is not r_0 must complete at time $< t_j$ (otherwise information could not continue onward to reach r_0 by time t_j). Therefore,

$$\{e_r : r \in A_j \setminus \{r_0\}\} \cap E(T_{\text{in}}) = \emptyset.$$

Hence the number of inter-group j-edges completed by time t_j satisfies

$$|E(H^{(j)}_{\le t_j})| \;\ge\; |E(T_{\text{in}})| + \big|\{e_r : r \in A_j \setminus \{r_0\}\}\big| \;\ge\; (q-1) + (a_j - 1).$$

Edges after t_j. For any group $v \notin A_j$, correctness implies it must eventually compute s_j. If v received no inter-group j-edge after t_j, its inter-group information about coordinate j would never change after t_j, so it could never become able to compute s_j, a contradiction. Thus each $v \notin A_j$ must be the head of at least one inter-group j-edge completing after t_j. These edges are distinct across different v, giving at least $q - a_j$ further edges.

Per-coordinate bound and summation. Let $F^{(j)}_{\text{inter}}$ be the number of inter-group element-transfers carrying coordinate j. Combining,

$$F^{(j)}_{\text{inter}} \;\ge\; \big((q-1) + (a_j - 1)\big) + (q - a_j) \;=\; 2(q-1).$$

Summing over $j \in [n]$ yields $F_{\text{inter}} \ge 2(q-1)n$. □

Lemma 11 (Inter-server NIC traffic)**.** $F_{\text{nic}} \ge 2(q-1)n$.

Proof. Apply Lemma 10 with the q servers as groups (supernodes). □

Lemma 12 (Slow-server incident NIC volume)**.** *Let $F^{\text{send}}_s := F^{\text{send}}_{\text{s}}$ be the total NIC traffic sent by S_s, and let F^{recv}_s be the total NIC traffic received by S_s. Then*

$$F^{\text{send}}_s \ge n, \qquad F^{\text{recv}}_s \ge n.$$

Proof. Fix a coordinate $j \in [n]$ and write $s_j = \sum_{i=1}^{p} x_{i,j}$.

If no inter-server transfer carrying coordinate j is ever sent from S_s, then all other servers' views (restricted to inter-server information) are independent of the inputs in S_s for coordinate j, so by indistinguishability they cannot uniquely determine s_j for all inputs, contradicting correctness. Hence for each j, at least one j-element must be sent from S_s, giving $F^{\text{send}}_s \ge n$.

Similarly, we have $F^{\text{recv}}_s \ge n$. □

Proof of Theorem 3. Decompose the directed NIC traffic as

$$F_{\text{nic}} = F_{\text{h}\to\text{h}} + F_{\text{h}\to s} + F_{s\to\text{h}}, \qquad F_{s\to\text{h}} = F^{\text{send}}_s, \quad F_{\text{h}\to s} = F^{\text{recv}}_s.$$

Let

$$F^{\text{send}}_s = x\,n, \qquad F^{\text{recv}}_s = y\,n,$$

so $x \ge 1$ and $y \ge 1$ by Lemma 12.

Slow-server NIC bottlenecks. Since S_s has aggregate NIC rate g/ℓ,

$$T \ge \frac{F^{\text{send}}_s}{g/\ell} = \frac{\ell x\, n}{g}. \tag{13}$$

Similarly,

$$T \ge \frac{F^{\text{recv}}_s}{g/\ell} = \frac{\ell y\, n}{g}. \tag{14}$$

Healthy-server NIC bottlenecks. Ignoring the extra time cost of healthy–slow transfers on the healthy side, the total *sending* work across the $q-1$ healthy servers is at least $F_{\mathrm{h}\to\mathrm{h}} + F_{\mathrm{h}\to s}$, so

$$T \geq \frac{F_{\mathrm{h}\to\mathrm{h}} + F_{\mathrm{h}\to s}}{(q-1)g} = \frac{F_{\mathrm{nic}} - F_{s\to\mathrm{h}}}{(q-1)g}.$$

Using Lemma 11 and substituting $F_{s\to\mathrm{h}} = xn$ gives

$$\frac{T}{n} \geq \frac{1}{g}\left(2 - \frac{x}{q-1}\right). \tag{15}$$

Similarly, the total *receiving* work across healthy servers is at least $F_{\mathrm{h}\to\mathrm{h}} + F_{s\to\mathrm{h}}$, hence

$$\frac{T}{n} \geq \frac{1}{g}\left(2 - \frac{y}{q-1}\right). \tag{16}$$

Min–max and symmetry. Combining (13), (14), (15), and (16), for all $x, y \geq 1$,

$$\frac{T}{n} \geq \frac{1}{g}\max\left\{2 - \frac{x}{q-1},\ 2 - \frac{y}{q-1},\ \ell x,\ \ell y\right\}.$$

By symmetry in x, y (and convexity of the max of affine functions), the minimum over $x, y \geq 1$ is attained at $x = y = z \geq 1$, yielding

$$\frac{T}{n} \geq \frac{1}{g}\min_{z\geq 1}\max\left\{2 - \frac{z}{q-1},\ \ell z\right\}.$$

Solving the 1-D minimization. The two branches intersect at

$$2 - \frac{z}{q-1} = \ell z,$$

namely

$$z^\star = \frac{2(q-1)}{1+\ell(q-1)}.$$

If $z^\star \geq 1$, equivalently $\ell \leq 2 - \frac{1}{q-1}$, then the minimum is attained at $z^\star$ and

$$\frac{T}{n} \geq \frac{\ell z^\star}{g} = \frac{1}{g}\cdot\frac{2\ell(q-1)}{1+\ell(q-1)}.$$

If $z^\star < 1$, equivalently $\ell > 2 - \frac{1}{q-1}$, then over the feasible region $z \geq 1$ the minimum is attained at $z = 1$, which gives

$$\frac{T}{n} \geq \frac{1}{g}\max\left\{2 - \frac{1}{q-1},\ \ell\right\} = \frac{\ell}{g}.$$

Therefore,

$$T \ \geq\ \frac{n}{g}\cdot\begin{cases}\dfrac{2\ell(q-1)}{1+\ell(q-1)}, & 1 < \ell \leq 2 - \dfrac{1}{q-1},\\[2ex] \ell, & \ell > 2 - \dfrac{1}{q-1}.\end{cases}$$

Equivalently,

$$T \ \geq\ \frac{n}{g}\cdot\max\left\{\frac{2\ell(q-1)}{1+\ell(q-1)},\ \ell\right\}.$$

□

Similarly, *Theorem 3 ignores the extra time cost of healthy–slow transfers on the healthy side. Accounting for this cost yields a strictly stronger lower bound, which is in fact tight; see Appendix E for the matching upper bound and discussion.*

Theorem 13 (Tight Lower Bound: Single Straggler, Multiple GPU per Server)**.** *Any correct AllReduce algorithm satisfies*

$$T \ \geq\ \frac{n}{g}\cdot\max\left\{\frac{2\ell(q-1)}{\ell(q-2)+2},\ \ell\right\}.$$

Proof of Theorem 13. We follow the notation introduced in the proof of Theorem 3. Thus,

$$F_{\mathrm{nic}} = F_{\mathrm{h}\to\mathrm{h}} + F_{\mathrm{h}\to s} + F_{s\to\mathrm{h}}, \qquad F_{s\to\mathrm{h}} = F_s^{\mathrm{send}}, \quad F_{\mathrm{h}\to s} = F_s^{\mathrm{recv}},$$

and

$$F_s^{\mathrm{send}} = x\,n, \qquad F_s^{\mathrm{recv}} = y\,n,$$

with $x \geq 1$ and $y \geq 1$.

Slow-server NIC bottlenecks. Since S_s has aggregate NIC rate g/ℓ,

$$T \geq \frac{F_s^{\mathrm{send}}}{g/\ell} = \frac{\ell x\, n}{g}. \tag{17}$$

Similarly,

$$T \geq \frac{F_s^{\mathrm{recv}}}{g/\ell} = \frac{\ell y\, n}{g}. \tag{18}$$

Healthy-server NIC bottlenecks (time-weighted). Across the $q-1$ healthy servers, the aggregate send rate is $(q-1)g$. Sending healthy→healthy elements costs 1 time unit each, while any element sent to (or received from) S_s costs ℓ time units (the slow NIC throttles the transfer). Thus the total *sending* work across healthy servers is at least $F_{\mathrm{h}\to\mathrm{h}} + \ell F_{\mathrm{h}\to s}$, implying

$$T \geq \frac{F_{\mathrm{h}\to\mathrm{h}} + \ell F_{\mathrm{h}\to s}}{(q-1)g} = \frac{F_{\mathrm{nic}} - F_{s\to\mathrm{h}} + (\ell-1)F_{\mathrm{h}\to s}}{(q-1)g}.$$

Using Lemma 11 and substituting $F_{s\to\mathrm{h}} = xn$, $F_{\mathrm{h}\to s} = yn$ gives

$$\frac{T}{n} \geq \frac{1}{g}\left(2 + \frac{(\ell-1)y - x}{q-1}\right). \tag{19}$$

Similarly, the total *receiving* work across healthy servers is at least $F_{\mathrm{h}\to\mathrm{h}} + \ell F_{s\to\mathrm{h}}$, hence

$$\frac{T}{n} \geq \frac{1}{g}\left(2 + \frac{(\ell-1)x - y}{q-1}\right). \tag{20}$$

Min–max and symmetry. Combining (17), (18), (19), (20), for all $x, y \geq 1$,

$$\frac{T}{n} \geq \frac{1}{g}\max\left\{2 + \frac{(\ell-1)y - x}{q-1},\ 2 + \frac{(\ell-1)x - y}{q-1},\ \ell x,\ \ell y\right\}.$$

By symmetry in x, y (and convexity of the max of affine functions), the minimum over $x, y \geq 1$ is attained at $x = y = z \geq 1$, yielding

$$\frac{T}{n} \geq \frac{1}{g}\min_{z\geq 1}\max\left\{2 + \frac{(\ell-2)z}{q-1},\ \ell z\right\}.$$

Solving the 1-D minimization. If $\ell \geq 2$, both branches are nondecreasing in z, so the minimum is at $z = 1$ and $\frac{T}{n} \geq \frac{\ell}{g}$. If $1 < \ell \leq 2$, the minimum is attained at the intersection

$$2 + \frac{(\ell-2)z}{q-1} = \ell z,$$

i.e.,

$$z^\star = \frac{2(q-1)}{\ell(q-2)+2}, \qquad \frac{T}{n} \geq \frac{\ell z^\star}{g} = \frac{1}{g}\cdot\frac{2\ell(q-1)}{\ell(q-2)+2}.$$

Therefore,

$$T \ \geq\ \frac{n}{g}\cdot\begin{cases}\ell, & \ell \geq 2,\\ \dfrac{2\ell(q-1)}{\ell(q-2)+2}, & 1 < \ell \leq 2.\end{cases}$$

Equivalently,

$$T \ \geq\ \frac{n}{g}\cdot\max\left\{\frac{2\ell(q-1)}{\ell(q-2)+2},\ \ell\right\}.$$

□

Remark 14 (Dropping the NVLink term)**.** *In addition to the NIC constraints, we also have the NVLink-time constraint*

$$T \;\geq\; \frac{F_{\mathrm{nv}}}{qv},$$

since the q servers have total aggregate NVLink capacity qv. Using the global traffic bound $F \geq 2(p-1)n$ and $F_{\mathrm{nv}} = F - F_{\mathrm{nic}}$, we get

$$F_{\mathrm{nv}} \;\geq\; 2(p-1)n - F_{\mathrm{nic}}.$$

Plugging in the minimum possible inter-server traffic $F_{\mathrm{nic}} = 2(q-1)n$ (from Lemma 10) yields

$$T \;\geq\; \frac{2(p-1)n - 2(q-1)n}{qv} \;=\; \frac{2(p-q)n}{qv} \;=\; \frac{2(g-1)n}{v},$$

where we used $p = gq$. Under our NVLink-rich assumption $v \geq g(g-1)$, this term is at most $2n/g$, and hence it does not affect the main results of this section.

Intuitively, in NVLink-poor *settings (e.g., per-GPU NVLink bandwidth is* not *at least $(g-1)$ times the NIC bandwidth, i.e., $v < g(g-1)$ in our normalization), then $F_{\mathrm{nv}}/(qv)$ can become the dominant bottleneck and should be kept explicitly. Conversely, when NVLink is at least this $(g-1)$-fold "multiplier," intra-server aggregation is never the limiter.*

C Time Analysis for Bubble Filling ($\ell < 2$)

When $\ell < 2$, each Stage 2/3 flow occupies $\ell\, s$ time on the straggler link while the corresponding ring slot takes $2s$. Bubble filling enlarges every Stage 2/3 flow to carry an additional $(2{-}\ell)\, s'/\ell$ elements of a point-to-point AllReduce between the healthy servers and the straggler, where s' denotes the new section size. The enlarged flow has duration $(1 + (2{-}\ell)/\ell) \cdot \ell\, s' = 2s'$, matching the ring slot exactly.

Data accounting. Each group of four segments (one per pattern) processes two kinds of data:

- **Ring path:** $(p{-}1)$ sections of size s', totalling $(p{-}1)\, s'$ elements.
- **P2P path:** $(2{-}\ell)\, s'/\ell$ additional elements allreduced via the straggler bubbles.

With $k/4$ groups (i.e. k segments total) the constraint $n = k\big[(p{-}1)\, s' + (2{-}\ell)\, s'/\ell\big]$ gives

$$s' \;=\; \frac{n\,\ell}{k\big[(p{-}2)\ell + 2\big]}\,.$$

Total time. Because every slot now has duration $2s'$, each middle body takes $2(p{-}1)\, s'$. The straggler's serialized recv chain across all $k{+}1$ bodies takes $2k(p{-}1)\, s'$ (the HEAD bub-up and TAIL Stage 2 flows are shorter than middle Stage 2 flows, but their durations sum to exactly $2k(p{-}1)\, s'$). In addition, ring-flow dependencies at the HEAD and TAIL add a startup/drain overhead of $2(\ell{-}1)(p{-}1)\, s'$ (at $\ell = 1$ the HEAD/TAIL flows are short enough to hide entirely within the straggler chain; at $\ell = 2$ they contribute a full extra body). Hence

$$T = \big[2k + 2(\ell{-}1)\big]\,(p{-}1)\, s' = \frac{2(p{-}1)\,\ell\, n}{(p{-}2)\ell + 2}\,\frac{k + \ell - 1}{k}$$
$$\xrightarrow{k\to\infty}\; \frac{2(p{-}1)\,\ell\, n}{(p{-}2)\ell + 2}\,.$$

Comparing with Theorem 6, we see that our algorithm *achieves bandwidth optimality without requiring $p \to \infty$*, confirming its optimality even for small clusters.

Overhead bound. Compared with the non-degraded baseline $T_0 = 2(p{-}1)\, n/p$ (single GPU per server, $g{=}1$),

$$\frac{T}{T_0} = \frac{p\,\ell}{(p{-}2)\ell + 2}\,.$$

This ratio is increasing in ℓ on $[1, 2]$ and attains its maximum $p/(p{-}1) = 1 + 1/(p{-}1)$ at $\ell = 2$, so the overhead is at most $1 + 1/(p{-}1)$ for all $\ell \in [1, 2]$.

D Algorithm Construction for Multiple Stragglers

We extend the four-stage pipelined algorithm of Section 4 from one straggler to $m \geq 2$ stragglers. Throughout, p denotes the total number of GPUs (one per server), of which GPUs $0, \ldots, m{-}1$ are stragglers with slowdown factors $\ell_1 \geq \cdots \geq \ell_m > 1$, and GPUs $m, \ldots, p{-}1$ are healthy.

D.1 Four-Stage Decomposition

As in Section 4.1, we partition the n input elements into k segments, each containing $p{-}m$ sections of size $s = n/(k(p{-}m))$. Each segment undergoes four stages:

1. **Stage 1 (Reduce-Scatter):** The $p{-}m$ healthy GPUs perform a reduce-scatter of the segment along a directed ring, accumulating partial sums over $p{-}m{-}1$ hops.
2. **Stage 2 (Upload):** Each healthy GPU sends its partial sum to *every* straggler. Each straggler receives $p{-}m$ uploads over its slow NIC.
3. **Stage 3 (Download):** Each straggler folds in its local contribution and sends the result back to every healthy GPU. Each straggler sends $p{-}m$ downloads.
4. **Stage 4 (Allgather):** The healthy GPUs allgather the global sums along the ring in $p{-}m{-}1$ hops.

The same two stage orderings from Section 4.1 remain valid, and the *disjoint-resource* principle still holds: Stages 2/3 use only the straggler NICs while Stages 1/4 use only the healthy ring links.

D.2 Schedule Construction

We design four patterns—B, D, A′, C′—analogous to the single-straggler patterns of Section 4.2, but adapted to multiple stragglers. We illustrate the construction for $p = 7$, $m = 2$ (GPUs 0–1 are stragglers; GPUs 2–6 are healthy).

Window structure. Each parallel body is divided into $p{-}1$ windows, each comprising two sub-slots. Of these, $p{-}m{-}1$ windows carry merged ring flows (S1+S4 or S1+S3, as in the single-straggler case), and m windows carry straggler communication (one per straggler). The body duration is $2(p{-}1)\, s$ when all straggler flows fit inside a $2s$-wide window, which holds whenever $\ell_i \leq 2$ for all i.

Four patterns. The patterns are defined as follows:

- **Pattern B**: S3 → S1 → S4 → S2.
- **Pattern D**: S3 → S1 → S4 → S2 (same ordering as B, with the same offset).
- **Pattern A′**: S1 → S2 → S3 → S4.
- **Pattern C′**: S1 → S2 → S3 → S4 (same ordering as A′, with flows in Stages 1 and 3 shifted).

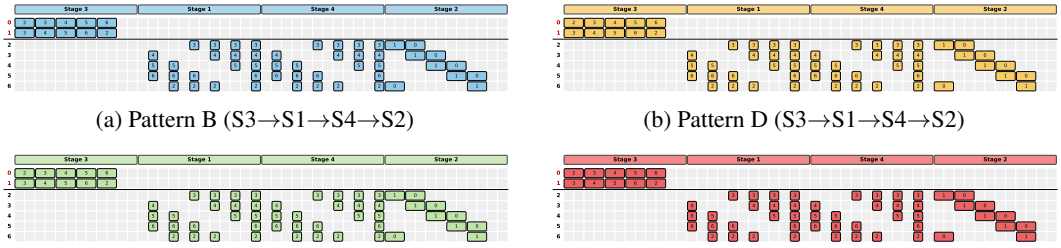
Figure 11 shows the four individual patterns for $p = 7$, $m = 2$.

Pipelining. Overlaying the four patterns produces the composite schedule of Figure 12. With k a multiple of 4, each pattern processes $k/4$ segments. Every parallel body in the steady state contains one instance of each stage from each pattern, fully utilizing both the healthy ring and all straggler links.

D.3 Time Analysis

Each segment has $p{-}m$ sections of size $s = n/(k(p{-}m))$. In each parallel body, the healthy GPUs occupy $p{-}1$ windows of $2s$ each, for a total of $2(p{-}1)\, s$. Each straggler i must send or receive $p{-}m$ flows, each of duration $\ell_i\, s$, giving a straggler chain of $\ell_i(p{-}m)\, s$. In the current schedule, additional $2(m-1)$ sections between healthy servers will be sent after the healthy servers finish communication with stragglers. The body duration is therefore

$$T_{\text{body}} \;=\; \max\{\, 2(p{-}1)\, s,\;\; (\ell_1(p{-}m) + 2(m-1))\; s \,\}\;.$$



(a) Pattern B (S3→S1→S4→S2)

(b) Pattern D (S3→S1→S4→S2)

(c) Pattern A′ (S1→S2→S3→S4)

(d) Pattern C′ (S1→S2→S3→S4)

Figure 11: Flow schedules for the four multi-straggler patterns (p=7, m=2). Rows 0–1 (red labels) are stragglers; rows 2–6 are healthy. Each cell's label indicates the destination GPU. Colored cells belong to the named pattern; uncolored cells belong to other patterns.

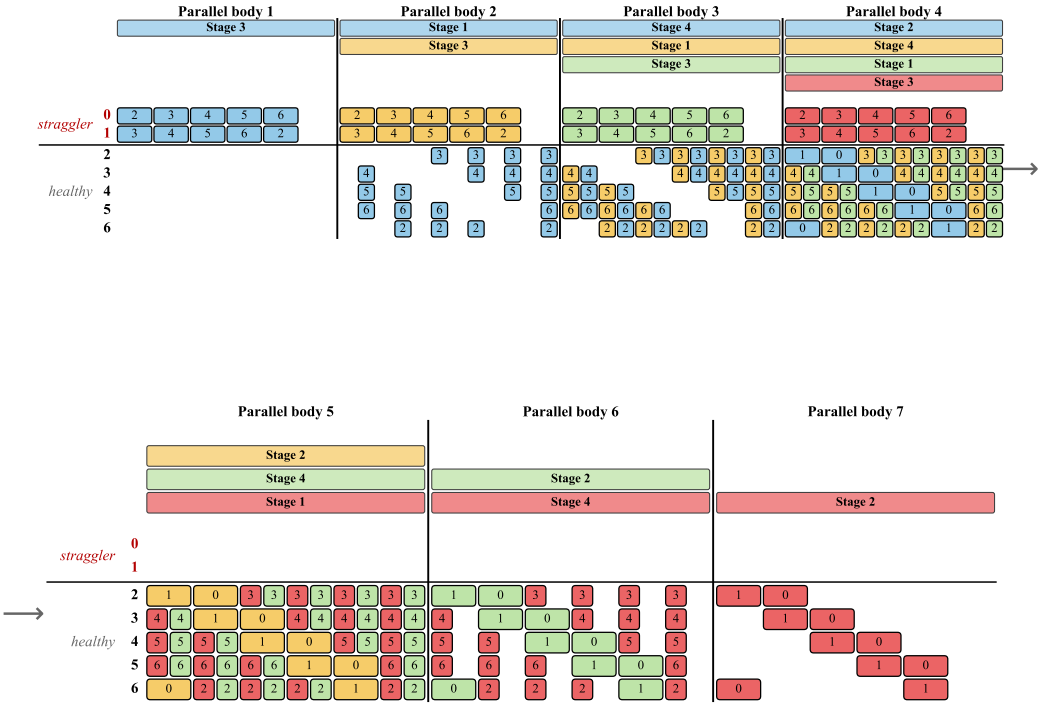


Figure 12: Combining the four multi-straggler patterns into a complete pipeline (p=7, m=2). Rows 0–1 are stragglers; rows 2–6 are healthy. The schedule has 7 parallel bodies; each steady-state body interleaves all four stages across the four patterns. No NIC sends or receives two flows simultaneously.

With $k/4$ groups (each spanning four patterns) plus startup/drain equivalent to one body, the total time is

$$T \;=\; \left(\frac{k}{4}+1\right)\cdot 4\,T_{\text{body}} \;=\; (k+4)\,T_{\text{body}}\,.$$

Case $\ell_1 < 2$ (ring-bottlenecked). The body is determined by the healthy GPUs: $T_{\text{body}} = 2(p{-}1)\,s$. Substituting s:

$$T = (k+4)\cdot 2(p{-}1)\,s = \frac{2(p{-}1)}{p{-}m}\,n\,\frac{k+4}{k}$$
$$\xrightarrow{k\to\infty}\; \frac{2(p{-}1)}{p{-}m}\,n\,.$$

Case $\ell_1 \geq 2$ (straggler-bottlenecked). The body is determined by the worst straggler: $T_{\text{body}} = (\ell_1(p-m) + 2(m-1))\ s$. Substituting:

$$T = (k+4) \cdot (\ell_1(p-m) + 2(m-1))\ s = \frac{\ell_1(p-m) + 2(m-1)}{p-m}\, n\, \frac{k+4}{k}$$
$$\xrightarrow{k\to\infty} \left(\ell_1 + \frac{2(m-1)}{p-m}\right) n\,.$$

Comparison with the lower bound. From Theorem 2, the lower bound (for $p \gg m$) is $T \geq \max\{2(p-1)/(p-m+\sum 1/\ell_i),\ \ell_1\} \cdot n$. Our algorithm achieves

$$T \xrightarrow{k\to\infty} \max\left\{\frac{2(p-1)}{p-m},\ \ell_1 + \frac{2(m-1)}{p-m}\right\} \cdot n.$$

In both cases, the ratio of the achieved time to the lower bound approaches 1 as $p \to \infty$ with m fixed (or as $m/p \to 0$), confirming near-optimality in large clusters.

E Algorithm Design for Multi-GPU Servers

As outlined in Section 4.4, we now consider the practical setting where each server hosts multiple GPUs. Let p be the total number of GPUs, g the number of GPUs per server (typically $g \in \{1, 2, 4, 8\}$), and $q = p/g$ the total number of servers. Each GPU within a server has its own dedicated NIC for inter-server communication, and the g GPUs are additionally connected to each other through a high-bandwidth NVLink fabric.

Notation. We adopt the same symbols as Section 4. The total input of n elements is first split into g *parts* of size n/g, one per leading-GPU position; the cycle anchored at each leading GPU operates on its own part. Each part is then split into k *segments*, and each segment into $q-1$ *sections* of size

$$s \;=\; \frac{n}{g\,k\,(q-1)}$$

elements. A healthy NIC therefore takes s time units to transmit one section, and an NVLink hop on one local ring (running at $(g-1)\times$ NIC speed) takes $s/(g-1)$ time. The pattern-overlay construction of Section E.2 requires $8 \mid k$ (so that the four patterns each receive an integer number of segments) and $k \geq 24$ (so that the head, the steady-state interior, and the tail of the schedule do not overlap); we assume both throughout. Throughout this appendix we adopt the following bandwidth model for the NVLink fabric, which reflects the configuration commonly deployed in practice:

- Each GPU has its own pair of NVLink up/down links, with per-direction bandwidth equal to $(g-1)$ times the NIC bandwidth, enough to keep one GPU communicating with all of its $g-1$ siblings at full NIC speed simultaneously.
- Different GPUs' NVLink lanes do not interfere, so all g local GPUs can send and receive at $(g-1)\times$ the NIC speed in parallel.

In particular, when one GPU broadcasts (or reduces) a value across the remaining $g-1$ GPUs, the operation completes in the same time that a single healthy NIC flow of equal size would take. At $g = 1$ the NVLink term disappears and the model degenerates to the single-GPU case of Section 4. Thanks to PXN NVIDIA Corporation (2022) (Section 2), all g NICs on a degraded server pool their bandwidth equally, so we may assume without loss of generality that they share the same slowdown factor ℓ.

We extend our algorithm to this setting in a way that closely parallels the classical AllReduce with NVLink: the data circulates over NVLink inside each server in addition to crossing the inter-server NIC ring. Our scheduler then overlaps the NVLink work with the NIC work so that the intra-server traffic does not extend the critical path.

The cleanest mental model is to run g concurrent copies of the single-GPU schedule, one per local GPU index. Inside every server the g GPUs are connected as an NVLink ring; each ring instance is "led" by a different local GPU position, and the g rings operate in parallel without interfering with

the inter-server schedule. The leading GPU of each ring uses that ring to collect its server's data on NVLink before each NIC send and to disperse the received data on NVLink after each NIC receive. Figure 13 sketches the topology: each server holds g GPUs that talk to each other over NVLink, and each GPU uses its own NIC to communicate with the corresponding GPU on every other server, so the leading GPUs across servers form g independent inter-server rings.

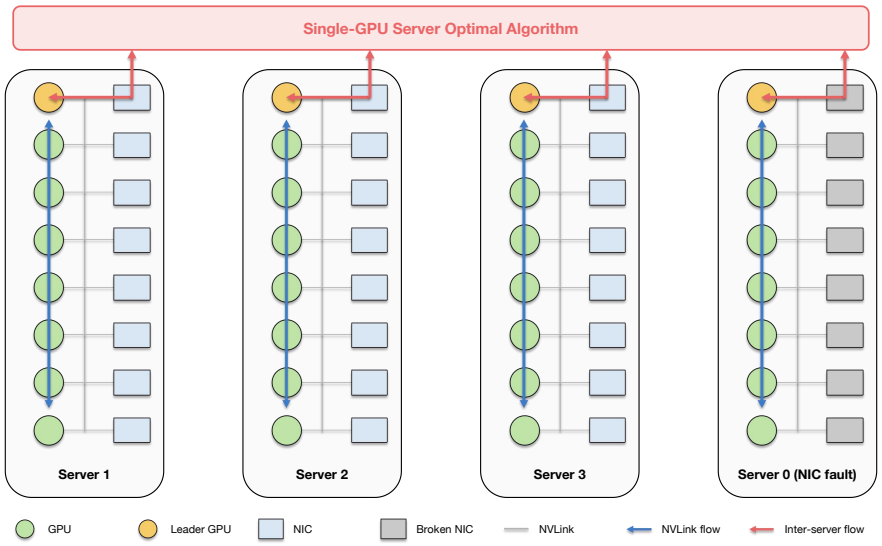


Figure 13: Multi-GPU servers: GPUs inside the same server communicate over NVLink, while GPUs on different servers communicate over the NIC. Each server hosts g GPUs and each GPU has its own NIC for inter-server traffic. We run g instances of the single-GPU schedule in parallel, anchored at each local GPU index in turn; intra-server NVLink transfers aggregate data before each NIC send and disperse it after each NIC receive. Each NVLink lane is provisioned at $(g-1)\times$ NIC bandwidth and operates independently per GPU and per direction.

E.1 NVLink Phases Around the Four Stages

Each of the four stages of Section 4.1 now requires NVLink work in addition to its NIC traffic. Before a section is sent out over the NIC, the g local GPUs must *collect* it on NVLink so the leading GPU holds the value to be transmitted; after a section arrives on the NIC, the leading GPU must *distribute* it to the other $g-1$ GPUs over NVLink. We rename the four original inter-server stages **S1, S2, S3, S4** and label the new intra-server stages **N1, N2, N3, N4**: each Sk is paired with a matching Nk placed either immediately before or immediately after it, and the linear dependencies among these N and S stages admit the two possible orderings shown in Figure 14. An N stage uses only NVLink and an S stage uses only the NIC.

The role of each N phase mirrors the corresponding S phase, and the ordering N $\leftrightarrow$ S is fixed by data dependencies:

- **N1** (collect): before S1, each healthy server reduces its local segment over NVLink onto the leading GPU, so that the leading GPU holds the value about to be sent over the NIC.
- **N2** (distribute): after S2, the straggler's leading GPU passes the just-received partial sum over NVLink to the other $g-1$ local GPUs. Under Schedule type 2 this is a plain broadcast; under Schedule type 1 the data instead traverses the local NVLink ring in one direction, with each GPU folding in its own contribution as it passes through, so that the GPU at the end of the ring accumulates the full partial sum.
- **N3** (collect): before S3, the straggler's leading GPU obtains over NVLink the value about to be sent over the NIC. Under Schedule type 2 this is a plain reduce of the local copies onto the leader; under Schedule type 1 the accumulated partial sum left at the end of the N2 ring traverses the

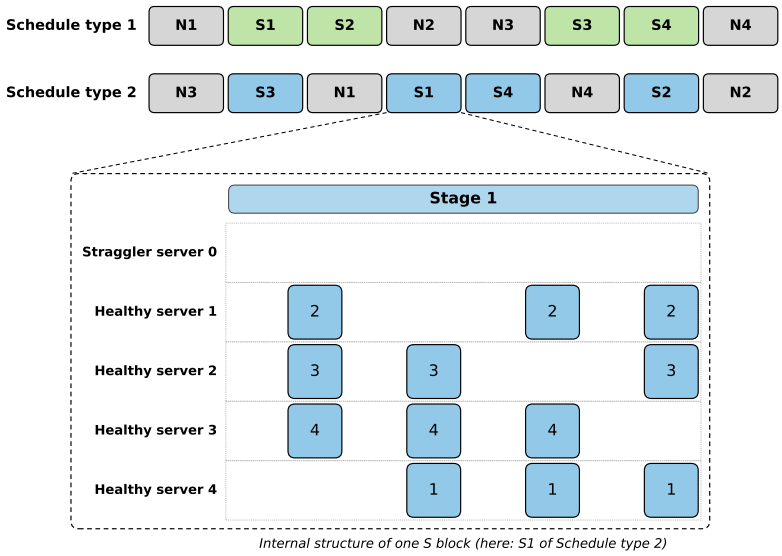


Figure 14: Schedule schematic showing the two possible orderings of NVLink (N) and NIC (S) stages that augment the four-stage decomposition of Section 4.1. The two schedule types correspond to the two stage orderings discussed therein.

same ring in the reverse direction, depositing the value onto every local GPU along the way before arriving at the leading GPU.

- **N4** (distribute): after S4, each healthy server's leading GPU broadcasts the just-received global sum over NVLink to the other $g-1$ local GPUs.

Data dependencies constrain the order of the eight phases within one loop iteration; Figure 14 shows two viable orderings.

E.2 Alternating N and S

Schedule types in Figure 14 do not parallelise cleanly across the four patterns: in some places two S phases sit next to each other while in others a single S is separated from the next S by an N, so no cyclic shift of the eight-phase sequence can align the four patterns into columns containing four distinct S phases or four N phases. To obtain such an alignment we defer selected phases to a later loop iteration, carrying along every phase that depends on them so that the relative order required by the data dependencies is preserved.

This gives the strict *N–S alternation* of Figure 15. Patterns A and C use the Schedule-type-1 sequence and patterns B and D use the Schedule-type-2 sequence. The figure has eight rows: rows 1–4 list patterns A, B, C, D once each, and rows 5–8 repeat the same four patterns shifted by one column, so that the eight rows together fill every NIC and every NVLink slot. Reading only the S phases of the first four rows recovers the order and overlay of the four single-GPU patterns in Figure 6. A primed label (e.g., N3′) marks a phase whose data comes from the *previous* loop iteration; a doubly-primed label (e.g., N4″) marks data from two iterations earlier.

Inside a single N body the intra-server work itself runs as a small NVLink ring across the g local GPUs. Figure 16 illustrates the traffic for Pattern A: the healthy and straggler servers each contribute an g-row panel, and each cell labels the destination GPU on the local ring. (We set $g = 4$ in the figure for illustration and label the four GPUs I/II/III/IV; the construction generalises to arbitrary g.) The collect phases (N1, N3) and the distribute phases (N2, N4) run the NVLink ring in opposite directions: collect uses the backward ring IV $\to$ III $\to$ II $\to$ I (leader receives, never sends), and distribute uses the forward ring I $\to$ II $\to$ III $\to$ IV (leader sends first). Combining the data flows from all g choices of leading GPU exactly fills the ring: at every sub-slot every GPU is sending to its ring neighbour, so all g NVLink links are saturated in parallel. Because NVLink is a factor of $g-1$

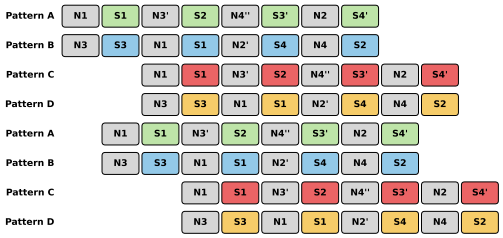


Figure 15: Eight-row N–S schedule built from schedule types of Figure 14. A primed label (e.g., N3′) marks a phase whose data comes from the previous iteration; a doubly-primed label (e.g., N4″) marks data from two iterations earlier.

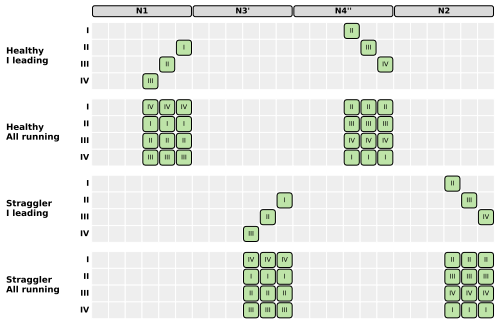


Figure 16: NVLink data flow, shown for Pattern A (other patterns follow the same template). Rows index the local GPUs on a server ($g = 4$). Each cell is one NVLink hop, labelled by the destination GPU; one hop carries $(p-1)\,s$ elements.

faster than the NIC, and one pattern inside a GPU-led cycle carries $(q-1)\,s = n/(gk)$ elements, the NVLink ring can cycle the data of all g leadings to every local GPU in just s time units, i.e. half a body, as shown in Figure 16.

Putting the four patterns together, both transports (NIC and NVLink) and both server types (healthy and straggler) combine into the composite schedule of Figure 17. The Inter-server panel is identical to the single-GPU composite of Section 4.2 (Figure 6); the two NVLink panels show how the local rings at each server type are kept busy in parallel. When $\ell = 2$, joining body 1 to body 9 turns Figure 17 into a closed cycle in which the NVLink bandwidth saturates in every odd body and the NIC bandwidth saturates in every even body; overlaying a second copy of the cycle shifted by one body then keeps both transports saturated at every instant.

E.3 Tail Optimization

Since the primed phases cross iteration boundaries (N3′, S3′, S4′ reach back one iteration and N4″ two), the schedule is not yet periodic at its endpoints. At the head, the first cycle (first 8 bodies of each pattern as shown in Figure 17) must leave every primed and doubly-primed phase empty and the second cycle must leave every doubly-primed phase empty, giving a head of 16 bodies; one can check that no further compression is possible. From body 17 onward every parallel body is filled. At the tail, the matching leftovers — one cycle that runs only the primed and doubly-primed phases, plus one cycle that runs only the doubly-primed ones — can be packed much more tightly: by merging remaining phases whenever the two can share a parallel body (e.g., a body labelled "S4/N4" carries both a Stage-4 NIC flow and the matching N4 step at the same time), the entire tail fits in at most 6 bodies, as shown in Figure 18. The last tail body is physically shorter than the others because its remaining inter-server work — a sole S4 phase of duration $s(q-2)/(q-1)$ — takes less than half of a normal $2s$ body.

This design applies to any value of ℓ, and the bottleneck is set by which resource saturates first. For $\ell > 2$ the straggler NIC is the bottleneck and the healthy NICs and both NVLink rings still have spare capacity; for $\ell < 2$ only the straggler NIC is under-utilised, while every healthy NIC and both NVLink rings run fully loaded.

E.4 Time Analysis

Recall that one healthy NIC hop on a section of s elements takes s time units. Under our bandwidth model each NVLink lane runs at $(g-1)\times$ NIC speed, so a single NVLink hop on one local ring takes $s/(g-1)$ time. Each N body actually performs *two* ring traversals — one collect and one distribute — for a total of $2(g-1)$ hops, costing $2(g-1)\cdot s/(g-1) = 2s$. For $\ell \le 2$ this matches the $2s$ duration of the S body exactly, and for $\ell > 2$ it is strictly shorter than the ℓs-long S body, so within the

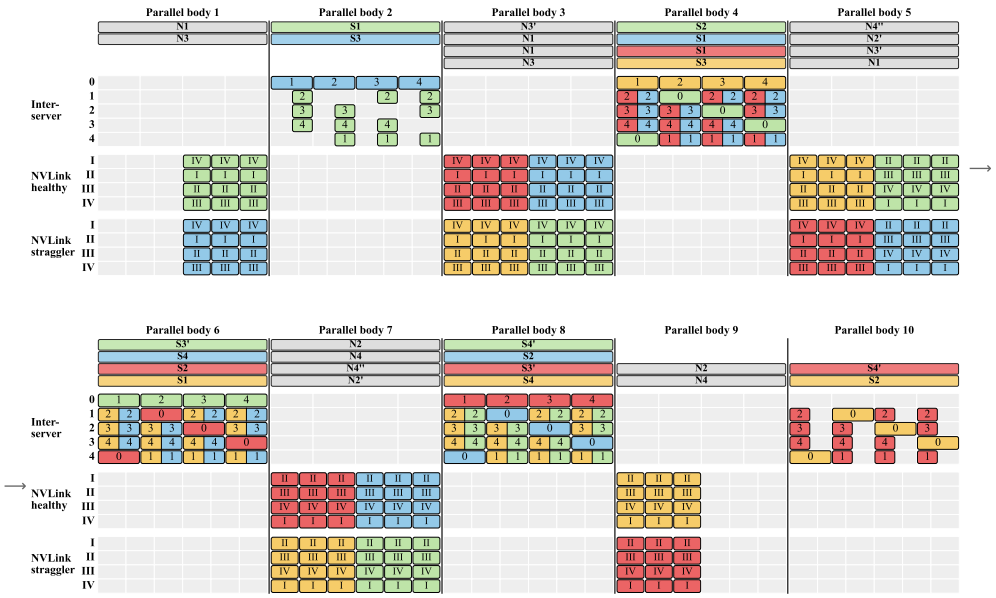


Figure 17: Composite multi-GPU schedule, drawn for $\ell = 2$. Each parallel body is either an N body (NVLink active, NIC idle) or an S body (NIC active, NVLink idle); N and S alternate strictly. The Inter-server table matches Figure 6; the NVLink healthy and NVLink straggler tables show how the local rings at each server type are filled. Closing this schedule into a cycle (body 1 glued to body 9) and overlaying a second copy shifted by one body fills every parallel body.

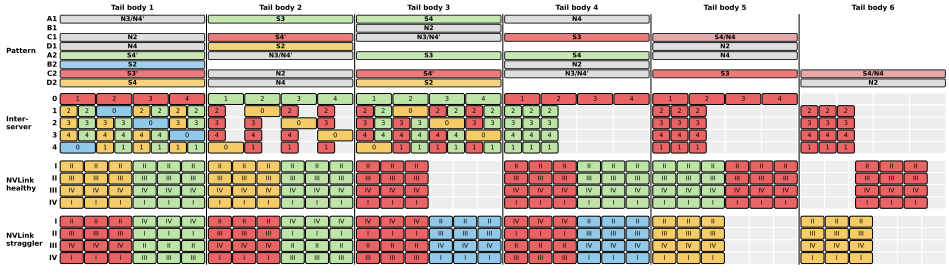


Figure 18: Tail schedule for the multi-GPU algorithm, drawn for $\ell = 2$. The eight stage-label rows (A1–D2) are exactly the four pattern rows of Figure 17 together with the same four rows shifted by one body — the two copies whose overlay fills the steady-state schedule. A label of the form "Sk/Nk" marks a tail body in which two phases are fused.

steady-state cycle the NVLink work always fits and adds no new critical path; the head and tail need a separate accounting.

The remaining accounting then follows the single-GPU analysis of Section 4.3 *without* the bubble-filling optimization; the bubble-filling subtlety is addressed in Case $\ell < 2$ below.

Counting bodies is now straightforward (using the head/tail decomposition of Section E.3). With k segments in total, the steady-state interior runs from body 17 through body k, for $k - 16$ fully filled bodies; the head spans the first 16 bodies but is effectively only 15.5 body-lengths since the first half of body 1 is empty; the tail occupies at most 6 bodies. The total schedule length is therefore at most

$$(k - 16) \;+\; 15.5 \;+\; 6 \;=\; k + 5.5$$

body-lengths.

Case $\ell \geq 2$ (straggler-bottlenecked). Each parallel body takes $\ell\,(q{-}1)\,s = \ell n/(gk)$ time. Multiplying by the body count $k + 5.5$ derived above,

$$T \;\leq\; \ell\,(q{-}1)\,s\,(k + 5.5) \;=\; \frac{\ell\,n\,(k + 5.5)}{g\,k} \;\xrightarrow{k\to\infty}\; \frac{\ell\,n}{g}\,, \tag{21}$$

i.e. the g leading-GPU cycles, each handling n/g elements via its own ring of NICs, jointly recover a $g\times$ speed-up over the single-GPU schedule of Equation (1). This matches the $\ell n/g$ lower bound from Theorem 3 and Theorem 13 in this regime, so the algorithm is *bandwidth optimal without requiring* $p \to \infty$.

Case $\ell < 2$ (ring-bottlenecked). Without bubble filling each body takes $2\,(q{-}1)\,s = 2n/(gk)$ time regardless of ℓ, giving

$$T \;\leq\; 2\,(q{-}1)\,s\,(k + 5.5) \;=\; \frac{2\,n\,(k + 5.5)}{g\,k} \;\xrightarrow{k\to\infty}\; \frac{2\,n}{g}\,. \tag{22}$$

Recall that Theorem 13 (strictly tighter than Theorem 3) gives the lower bound

$$T_{\mathrm{LB}} = \frac{n}{g} \cdot \frac{2\ell(q-1)}{\ell(q-2)+2}$$

for $\ell < 2$. The ratio of our $T = 2n/g$ to this bound is

$$\frac{T}{T_{\mathrm{LB}}} \;=\; \frac{\ell(q-2)+2}{\ell(q-1)} \;<\; \frac{q}{q-1}\,.$$

so the algorithm is within a factor $q/(q{-}1)$ of the theoretical lower bound across the whole $\ell < 2$ range. As $q \to \infty$, this factor approaches 1, confirming near-optimality in large clusters.

Bubble-filling in the multi-GPU server case. The bubble-filling technique of Section 4.2 cannot be directly applied here: for $\ell \leq 2$, each body already saturates the local NVLink ring, leaving no spare NVLink bandwidth to carry the additional P2P traffic that bubble filling would inject. In principle, one could slice each straggler bubble into g pieces and distribute them across the g leading-GPU cycles to recover that bandwidth. This would close the remaining $q/(q{-}1)$ gap and match the lower bound of Theorem 13 exactly without requiring $q \to \infty$ (following a time analysis similar to Appendix C); however, the resulting savings are marginal and do not justify the added scheduling complexity.

E.5 Potential Extension to Multiple Straggler Multiple GPU per Server Case

The alternating N-S phase construction here can be combined with the new S phase developed in Appendix D. The only obstacle is that patterns A and C are now replaced by patterns A' and C', which also alters the alternating N-S schedule. A modified schedule is provided in Figure 19. Due to space constraints, we omit the full construction details, time analysis, and corresponding theoretical lower bound.

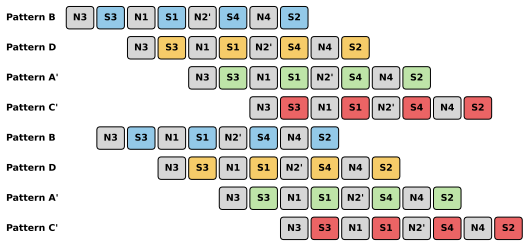


Figure 19: Eight-row alternating N–S schedule with patterns A′ and C′ substituted for patterns A and C.

F Choice of the Number of Pipeline Segments k

The single-straggler analysis (Section 4) shows that, ignoring per-message overhead, the pipeline head/tail overhead scales as $O(1/k)$, so larger k is always beneficial in the idealized cost model. In practice, however, per-message overhead is proportional to the total number of flows. In our algorithm, the total flow count is $O(kp^2)$, and each server sends $O(kp)$ flows, so the $O(k)$ growth in per-message cost eventually outweighs the $O(1/k)$ pipeline-overhead reduction. This trade-off creates a unique optimum that is analytically solvable given a measured per-flow launch cost.

We verify the trade-off empirically. The setup uses the MegaScale-175B configuration Jiang et al. (2024): 8,192 GPUs with TP=8, PP=8, DP=128. Because each gradient AllReduce runs within one data-parallel group, we set p=128. We model one representative gradient chunk of N=1 GB with a single degraded participant (m=1) at ℓ=2 (the affected NIC runs at 100 Gbps while healthy NICs run at 200 Gbps). We sweep $k \in \{7, 14, 21, 28, 42, 56, 84\}$, holding all other parameters fixed.

Table 5: Finite-k sweep for the MegaScale-175B configuration (p=128, ℓ=2, N=1 GB, m=1). Segment size s shrinks as k grows; OptCC completion time is U-shaped, with an empirical minimum at k=21.

k	7	14	21	28	42	56	84
Segment s (MB)	128	64	42.67	32	21.33	16	10.67
OptCC time (ms)	100.11	94.84	93.48	96.58	96.78	100.74	107.91
Theory fault-free T_0 (ms)	81.28	81.28	81.28	81.28	81.28	81.28	81.28

At k<21, head/tail overhead dominates and shrinking segments help; beyond k=21, the growing per-flow launch cost takes over. In deployment we pick k by minimizing this closed-form combined cost using a one-time offline profile of the target hardware's per-flow overhead.

G End-to-End Training-Iteration Overhead

To evaluate how much of OptCC's collective-level speedup translates into end-to-end training throughput, we simulate one training iteration of GPT-175B pretraining under a 50% endpoint-bandwidth loss. The setting follows the MegaScale-175B configuration Jiang et al. (2024): 8,192 A100 GPUs with TP=8, PP=8, DP=128; per-GPU micro-batch size 1; global batch size 6,144; sequence length 2048; 12,582,912 tokens per iteration. Healthy endpoints operate at 200 Gbps; one participant in the affected data-parallel group operates at 100 Gbps (ℓ=2).

Table 6: Simulated end-to-end GPT-175B iteration time under a 50% endpoint-bandwidth loss. Iteration-time overhead is normalized to $\text{NCCL}_{\text{NoFailure}}$.

Method	Iteration (s)	Throughput (Mtoken/s)	Iteration overhead
$\text{NCCL}_{\text{NoFailure}}$	8.607	1.462	0.00%
ICCL Zhang et al. (2025)	9.027	1.394	4.88%
R^2CCL Wang et al. (2025a)	8.817	1.427	2.44%
OptCC	8.646	1.455	0.45%

OptCC caps the failure-induced iteration overhead at 0.45%, compared with 4.88% for ICCL and 2.44% for R^2CCL—a $10.8\times$ reduction relative to ICCL and a $5.4\times$ reduction relative to R^2CCL in the failure-induced iteration-time overhead.