Breaking Temporal Symmetry in Collective Communication Algorithms

Arjun Devraj
Cornell University
Ithaca, NY, USA

Rachee Singh
Cornell University
Ithaca, NY, USA

Abstract

Distributed machine learning inherited collective communication algorithms from high-performance computing. The cost models behind these algorithms assume that every rank is ready to communicate exactly when the algorithm's schedule demands, an assumption we call *temporal symmetry*. In practice, however, ranks arrive at collectives hundreds of microseconds apart. Since collective algorithms impose strict data dependencies across ranks, this *temporal asymmetry* stalls the collective regardless of how it is implemented. In fact, we prove that bandwidth-optimal algorithms (e.g., Ring) are especially fragile to this asymmetry because their data transfer schedules require every rank to transmit in every round. We instead propose designing *asymmetry-resilient* collective algorithms that spend the arrival gap on useful communication among ready ranks, trading a modest increase in total communication for a much larger reduction in critical-path communication. Through the design of a new cost model and an expansive research agenda, we envision a future where collective algorithms are no longer static schedules developed for idealized distributed computation but adaptive artifacts shaped by the stochastic behavior of the systems they run on.

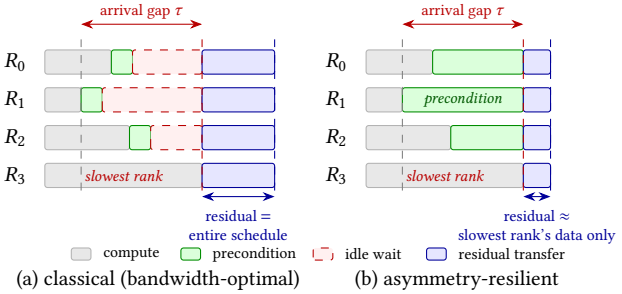
CCS Concepts

• **Computing methodologies** → **Parallel algorithms**; *Machine learning*; • **Networks** → **Network algorithms**; **Network performance modeling**.

Keywords

Collective communication, Distributed machine learning

ACM Reference Format:
Arjun Devraj and Rachee Singh. 2026. Breaking Temporal Symmetry in Collective Communication Algorithms. In *The 25th ACM Workshop on Hot Topics in Networks (HotNets '26), November 16–17, 2026, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3847290.3859955



(a) classical (bandwidth-optimal) (b) asymmetry-resilient

Figure 1: The slowest rank R_3 arrives at the collective τ μs after the fastest rank R_1. (a) In a bandwidth-optimal schedule, the *residual communication* (blue) after the slowest rank arrives is the entire schedule due to dependencies that require its data from the start. (b) An asymmetry-resilient algorithm instead uses the gap to bootstrap communication among faster ranks and then executes a short *completion* phase once the slowest rank arrives, reducing overall execution time.

This work is licensed under a Creative Commons Attribution 4.0 International License.
HotNets '26, Salt Lake City, UT, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-3029-0/2026/11
https://doi.org/10.1145/3847290.3859955

1 Introduction

Distributed machine learning (ML) workloads rely on collective communication primitives (e.g., AllReduce, AllGather) to exchange model state across a group of GPU ranks. Algorithms that implement these collectives, such as Ring [41] and Recursive Halving/Doubling [46], and their performance models originate from decades of research in high-performance computing (HPC). For example, the popular Hockney (α–β) model quantifies the time to send m bytes of data as $\alpha + \beta m$, where α is the fixed per-message latency and β is the inverse bandwidth of the network [19, 46].

Optimal algorithms under these models minimize either the latency or bandwidth costs of communication. Latency-optimal algorithms send few messages to incur the α cost infrequently while bandwidth-optimal algorithms minimize the bytes sent over the network to incur a small β cost. One key trick in collective algorithms is to schedule transfers between disjoint pairs of ranks in parallel, so one "round" of communication takes the wall-clock time of a single transfer while moving data between many pairs. In the popular Ring AllReduce algorithm, each rank sends an equal amount of

data to its neighbor in every round, achieving bandwidth optimality by transmitting the fewest number of bytes.

We revisit a key assumption in how we reason about collective algorithms. When the algorithm's schedule decides that two ranks should communicate, we assume both ranks are ready to do so. We call this assumption *temporal symmetry*, i.e., symmetry in the time at which ranks are ready to communicate. In practice, ranks become ready hundreds of microseconds apart due to variation in prior kernel runtimes [1, 13, 27, 31, 45]. We refer to this variation in rank arrivals at the collective as *temporal asymmetry*. Regardless of its cause, temporal asymmetry degrades the realized performance of theoretically optimal collective algorithms.

Performance degradation due to temporal asymmetry occurs regardless of how a collective algorithm is implemented. One common implementation places a *global barrier* before the collective so that communication only begins once all ranks are ready. In this design, early ranks idle until the slowest rank arrives at the barrier, delaying the entire collective. Libraries like NVIDIA's Collective Communication Library (NCCL) [38] adopt a more sophisticated approach that allows each rank to wait only on the peer it needs to communicate with in that round [20]. However, the slowest rank still delays the completion of the collective because the algorithm (e.g., Ring) imposes strict data dependencies between ranks (Figure 1a). Faster ranks can finish a small prefix of exchanges that do not involve a slow rank's data (green), but they then stall waiting on transitive data dependencies (red). As a result, the number of bytes that must be transferred after the last rank arrives, which we call the *residual communication* (blue), is identical in both implementations.

The common solution has been to engineer the asymmetry away by detecting and evicting slow ranks [12, 23, 60]. While these efforts mitigate the egregious delays of malfunctioning hardware/software, much of the asymmetry is routine and a reality of distributed computation [11, 27, 31, 45]. While it may be tempting to ignore these delays, at the FLOP rates of modern GPUs, even a microsecond of idling—let alone hundreds—forfeits over a billion floating-point operations per rank. Moreover, as inter-GPU interconnect bandwidth grows exponentially [39], temporal asymmetry could even rival the time spent transferring data during the collective. As a result, the delays too small for the infrastructure to eliminate have become too expensive for algorithms to ignore.

We envision a research agenda that stops treating temporal asymmetry as a defect to be engineered away and instead designs collective algorithms that adapt to it. We propose a class of *asymmetry-resilient* collective algorithms that allow faster ranks to begin communicating while waiting for slower ranks, folding in slower ranks' contributions as they arrive. Our design principle structures the schedule's data dependencies so that the later a rank arrives, the less of the schedule depends on its data. We show that such algorithms must deliberately relax bandwidth optimality, trading a modest increase in total communication for a much larger reduction in residual communication. Thus, a collective algorithm's cost depends on the interplay between its data transfer schedule, its dependency graph, and the stochastic process by which ranks become ready to communicate.

Designing this class of algorithms is challenging for several reasons. First, arrival times are stochastic runtime quantities: which ranks arrive late, by how much, and in what order are all unknown. A schedule fixed in advance must therefore improve *expected* performance while adding negligible overhead even when there is no gap in arrival times. Second, we show that bandwidth optimality is fundamentally at odds with asymmetry resilience (§2). A bandwidth-optimal schedule requires every rank to transmit an equal amount of data. However, the slowest rank can only begin sending its share once it arrives, serializing all of its transfers on the critical path (§3). Enabling *productive* communication during the arrival gap must therefore abandon the bandwidth-optimal schedules used today and navigate a rich design space that tackles both algorithmic and systems challenges (§4).

2 An Inheritance from HPC

Distributed ML has become a dominant workload in datacenters, yet it communicates using primitives and performance models designed decades ago for supercomputers [36].

Collective communication. A collective primitive executes an operation across a group of processes. These primitives have been curated by the Message Passing Interface (MPI) [36], used in supercomputers for decades, and they have been adopted essentially intact by distributed ML [30, 37, 38]. For example, the AllReduce primitive averages gradients in data parallelism and aggregates partial matrix products in tensor parallelism. ReduceScatter and AllGather compose into AllReduce and enable fully-sharded data parallelism (FSDP) while AllToAll routes tokens in expert parallelism [1]. Each primitive is implemented by an algorithm that specifies a schedule of point-to-point transfers over discrete *rounds* of communication. In a round, disjoint pairs of accelerators or *ranks* exchange data in parallel.

Cost model. Researchers analyze the performance of collective algorithms using the Hockney (α–β) model [19], the standard cost model for collective communication [44, 51, 55]. As per this model, the time to execute a collective is:

$$T = \sum_{r=1}^{R} \left(\alpha + \left(\max_i s_{r,i} \right) \beta \right) = R\alpha + \left(\sum_{r=1}^{R} \max_i s_{r,i} \right) \beta. \quad (1)$$

α and β are the infrastructure-dependent latency and bandwidth constants, R is the number of rounds in the collective algorithm, and $s_{r,i}$ is the number of bytes sent by rank i in round r. Since all $s_{r,i}$ in a round r are sent in parallel, each

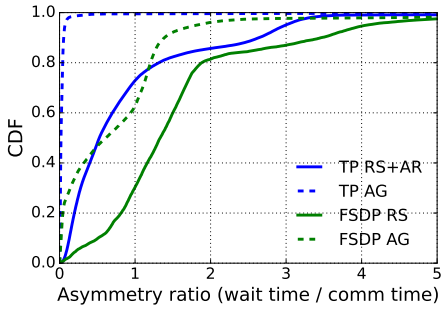


Figure 2: *Asymmetry ratio*—the time a collective spends waiting for its slowest rank divided by the time it spends transmitting data—from finetuning Llama-3 70B on NVIDIA DGX H100 and H200 nodes (8 GPUs each). The ratio for TP AllGather (AG) is lower because it directly follows the TP ReduceScatter (RS), i.e., no preceding computation.

round's duration is determined by the rank that sends the most bytes. We write this more compactly as $T = R\alpha + q\beta$, where $q = \sum_{r=1}^{R} \max_i s_{r,i}$ is the total bytes transmitted on the collective's *critical path*. While transfers within a round are parallelized, rounds execute sequentially. The critical path is therefore the chain of each round's largest transfer. Every byte on the critical path adds to the collective's total time.

Optimal collective algorithms. Latency-optimal algorithms minimize R while bandwidth-optimal algorithms minimize q. The large buffers in distributed ML make bandwidth-optimal algorithms (e.g., Ring) the standard [10, 20, 44]. Prior work has proven that the lower bound on bandwidth complexity for an AllReduce over n GPUs with total buffer size S is $q=\frac{2(n-1)}{n} S$, achieved by bandwidth-optimal algorithms such as Ring and Recursive Halving/Doubling [41].

2.1 Temporal Asymmetry in Collectives

The Hockney model assumes *temporal symmetry*, i.e., communicating ranks are simultaneously ready to transfer data and no rank has to wait for another. This assumption was safe when collectives were first proposed. Interconnects were slow, so the time to move a collective's data over the network dwarfed the overhead of temporal asymmetry. Reasoning about collective performance in terms of latency and bandwidth costs alone was accurate because the asymmetry was a negligible overhead. Today the balance has flipped. Inter-GPU link bandwidth is measured in gigabytes per second (e.g., 900 GB/s on NVIDIA B200s), which shrinks communication time and leaves temporal asymmetry as a significant fraction of end-to-end collective runtime. Meanwhile, ML scale-up domains have grown from 8 GPUs to 144 GPUs [40], which widens the spread of rank arrival times at the collective [13, 16].

Empirical evidence. While recent work has highlighted straggler delays in large-scale ML [31, 45], we analyze their impact on collective communication within a single multi-GPU server. We finetuned Llama-3 70B [17] in BF16 for 100 iterations on the NVIDIA DGX H100 and H200 (8 GPUs with 450 GB/s P2P bandwidth). We used the `torchtitan` ML framework [30] and applied 4-way tensor parallelism (TP) and 2-way fully sharded data parallelism (FSDP). The traces from these runs contained over 100K collective operations, at per-rank kernel-level granularity. Figure 2 shows that temporal asymmetry is a significant source of inefficiency in ML collectives. For ReduceScatter and AllReduce, the median wait time in the collective is 0.5× and 1.3× (5× at worst) its data transfer time for TP and FSDP, respectively, with typical delays in the tens to hundreds of microseconds.

> **Takeaway 1: Temporal symmetry no longer holds in practice. Ranks arrive at collectives far apart in time, and the wait now rivals the transfer time.**

Implementation does not matter. It is tempting to view temporal asymmetry as an artifact of how collective algorithms are implemented. The simplest implementation places a global barrier before the collective, so every rank waits until all ranks are ready before any data is sent. However, production libraries like NCCL forgo this global barrier. Instead, each rank spin-waits on a flag its neighbor sets and begins communicating the moment its inputs are available—a discipline called *lazy synchronization* [20]. Figure 3 shows why the impact of temporal asymmetry persists. A collective algorithm specifies a schedule of data transfers where each transfer depends on earlier ones. Every cell of the graph in Figure 3 (left) is a partial sum that can execute only when its operands exist. A cell whose running sum requires the slowest rank's x_3 *must wait* for that rank regardless of how the waiting is enforced. Lazy synchronization lets faster ranks complete a prefix of the schedule early, but the total communication cost is still as bad as before because *all* of the slowest rank's transfers lie on the critical path once it arrives.

Residual communication. We call the bytes that must be sent after the last rank arrives the *residual communication*, q_{res}. These bytes form the tail of the collective's critical path, and they can only move after the slowest rank arrives. Figure 3 shows how every round computes a cell that needs the slowest rank's x_3, so all of its transfers execute after it arrives; the residual communication spans the entire schedule. The collective thus pays the full arrival delay in addition to its entire communication cost. The early work of faster ranks (green cells) saves nothing on the critical path.

> **Takeaway 2: The cost of temporal asymmetry is not an implementation artifact, but due to fundamental algorithmic data dependencies that apply under barriers and lazy synchronization alike.**

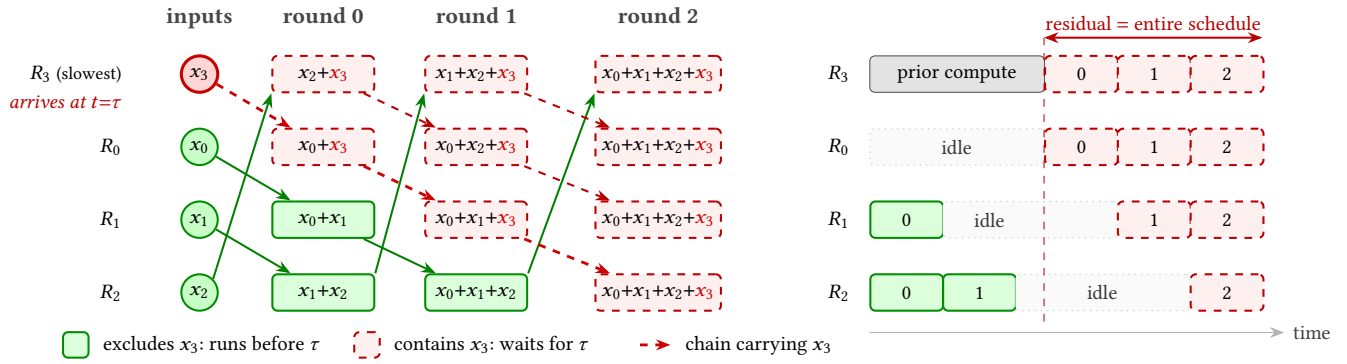


Figure 3: Ring ReduceScatter with n=4 ranks, where R_3 arrives the latest (at t=τ). Left: the schedule's dependency graph. Each cell is the partial sum a rank computes in that round, and each arrow shows a dependency on the transfers that produce the operands of the partial sum. A cell can complete before the arrival delay τ of R_3 (the slowest rank) if its partial sum excludes x_3. Even though early arriving ranks perform some useful computation (green cells) until R_3 arrives, every communication round needs R_3's data to compute a cell (x_2+x_3 in round 0, x_1+x_2+x_3 in round 1, etc.). This dependency forces the residual communication after R_3 arrives to incur the full delay of performing the collective, on top of the arrival delay τ. Right: timeline view of the schedule. Before τ, only R_1 and R_2 can start communicating, and R_0 cannot begin due to an immediate dependency on R_3; all three soon idle. Once R_3 arrives, every rank has to wait on its transfers to catch up, so the span after τ is the full q_{res}=q.

2.2 Residual vs. total communication

Bandwidth-optimal algorithms minimize q, i.e., serial bytes sent over the *entire* schedule. When all ranks initiate the collective simultaneously, these algorithms minimize the time spent on collective communication with large buffers. However, when ranks arrive at the collective at different times, q does not capture the collective's execution time (Figure 2). For some time, ranks stall waiting on other ranks to send the prerequisite data to begin the next round of the schedule. When accounting for this temporal asymmetry, we argue that the quantity that matters is instead the residual q_{res}, the bytes that must be sent once the last rank arrives. Minimizing q and minimizing q_{res}, as we show, are often incompatible objectives. For AllReduce, bandwidth-optimal algorithms (e.g., Ring) are widely adopted because they transmit the theoretical minimum number of bytes, i.e., $q=\frac{2(n-1)}{n}S$. However, we now show that these same algorithms *cannot* leverage temporal asymmetry to reduce the residual communication, i.e., q_{res}=q, making them brittle to temporal asymmetry.

Theorem 2.1. *If an AllReduce algorithm achieves the bandwidth optimal lower bound $q = 2\frac{n-1}{n}S$, then for every round r, all ranks must communicate the same number of bytes, i.e., $s_{r,1} = s_{r,2} = \cdots = s_{r,n}$. No rank can be idle in any round.*

Proof. Consider any AllReduce schedule with R rounds. Let $s_{r,i}$ be the total number of bytes transmitted by rank i in round r. Recall from Eq. 1 that the number of bytes transmitted on the critical path (i.e., bandwidth complexity) is $q=\sum_{r=1}^{R}\max_i s_{r,i}$. Let $V \triangleq \sum_{r=1}^{R}\sum_{i=1}^{n} s_{r,i}$ be the total number of bytes transmitted across all ranks over the schedule. Since for each round r, $\max_i s_{r,i} \geq \frac{1}{n}\sum_{i=1}^{n} s_{r,i}$, we can rewrite q as:

$$q = \sum_{r=1}^{R}\max_i s_{r,i} \geq \frac{1}{n}\sum_{r=1}^{R}\sum_{i=1}^{n} s_{r,i} = \frac{V}{n}. \quad (2)$$

Prior work has shown that any AllReduce of an S-byte buffer over n ranks must transmit at least $V \geq 2(n-1)S$ bytes [41]. Thus, the bandwidth lower bound is achieved with strict equality, when $q=\frac{2(n-1)}{n}S$, which requires equality in every round, i.e., $\max_i s_{r,i} = \frac{1}{n}\sum_i s_{r,i}$ for all r. This occurs only if $s_{r,1}$=$s_{r,2}$=$\cdots$=$s_{r,n}$ for every round r. In particular, no rank i can have $s_{1,i}$=0 in the first round unless *all* ranks have $s_{1,i}$=0 (i.e., a degenerate round with no data). Any schedule where some rank does not participate in the first round must have $q>\frac{2(n-1)}{n}S$, and cannot be bandwidth optimal. □

Minimizing q requires all ranks to send the same amount in every round, from the very start. This is because bandwidth optimality requires maximally parallelizing work to fully utilize total bandwidth in every round of the algorithm. Even with lazy synchronization, the slowest rank must still send $q=\frac{2(n-1)}{n}S$ bytes once it arrives under classical bandwidth optimality. When all ranks must transmit equally in every round, a delay at rank i serializes all R rounds behind it.

Corollary 2.2. *Let D_i be the critical-path bytes (portion of q) that cannot complete before rank i arrives. For any bandwidth-optimal AllReduce algorithm, D_i=q for* every *rank i.*

Takeaway 3: Bandwidth-optimal algorithms are *maximally fragile* to temporal asymmetry, motivating new algorithm design to minimize q_{res}.

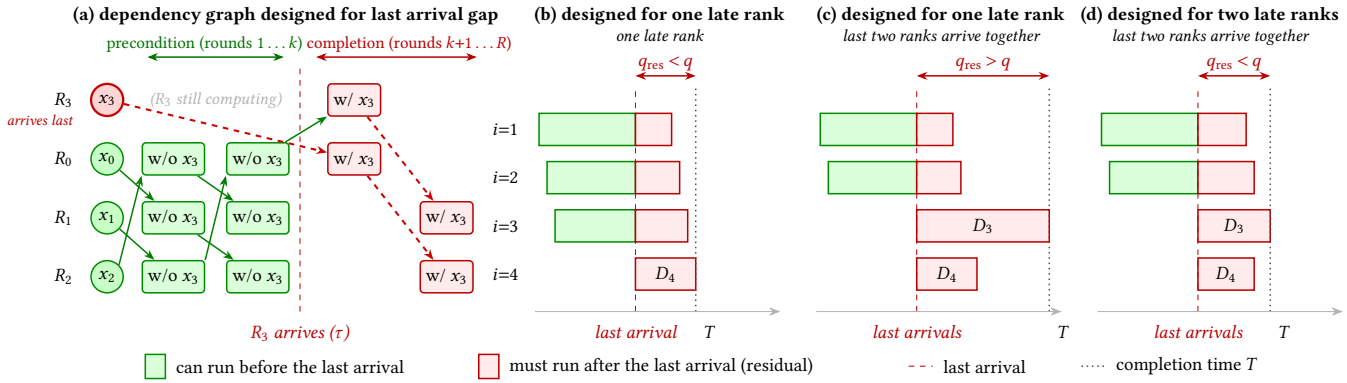


Figure 4: (a) AllReduce algorithm designed for the last arrival gap, generalized from recent work [14]. Unlike Figure 3, no cell in the first k rounds depends on x_3, allowing these cells to run ahead productively. (b)–(d) Dependence profiles that capture Eq. 3. Each row is a candidate critical path whose bar begins when the i-th rank arrives (τ_i) and extends for $D_i\beta$. While a profile designed solely for the last arrival gap can hide much of the schedule inside one large arrival gap (b), this performs poorly when the last *two* ranks arrive together (c), which requires designing an algorithm that targets a different arrival process (d).

3 Asymmetry-Resilient Algorithms

To reason about collective algorithms under temporal asymmetry, we define a cost model that takes the ranks' arrival times as input. We use $\tau_1 \leq \tau_2 \leq \cdots \leq \tau_n$ to denote the order statistics of the ranks' normalized arrival times (i.e., $\tau_1 = 0$) and henceforth index ranks by arrival order, where D_i denotes the bytes on the critical path (i.e., excluding parallel transfers in a round) that depend on the i-th arriving rank. Each arrival contributes a candidate critical path, as the D_i bytes of transfers that depend on rank i's arrival (Corollary 2.2) cannot execute before τ_i and thus cannot complete before $\tau_i + D_i\beta$. The collective only completes after the slowest among these finishes, so we model a collective algorithm's cost as:

$$T(\vec{\tau}) \;=\; \max_{1 \leq i \leq n} \left(\tau_i + D_i\,\beta\right), \tag{3}$$

excluding α terms that follow from Eq. 1. Note that $D_{i-1} \geq D_i$ for all i because whatever cannot complete before a later arrival by definition cannot complete before an earlier one.

Three special cases show that Eq. 3 captures existing algorithms and recovers today's cost model. First, two-phase algorithms that use the gap between the last two arrivals [14] (Figure 1b) run a precondition phase when the first $n-1$ ranks are ready, followed by a completion phase once the final rank arrives. These algorithms have $D_i = q_{\text{pre}} + q_{\text{res}}$ for $i < n$ and $D_n = q_{\text{res}}$, so Eq. 3 reduces to:

$$T(\vec{\tau}) = \max\left(\tau_{n-1} + q_{\text{pre}}\beta,\ \tau_n\right) + q_{\text{res}}\beta. \tag{4}$$

The completion phase can only begin after the precondition finishes and the slowest rank arrives; it then runs for $q_{\text{res}}\beta$. Second, for classical bandwidth-optimal algorithms, Corollary 2.2 shows that $D_i = q$ for all i, so $T(\vec{\tau}) = \tau_n + q\beta$ regardless of the shape of $\vec{\tau}$ (since τ_n is by definition the latest arrival). This reflects the inability of these algorithms to adapt to temporal asymmetry. Third, with perfect temporal symmetry ($\vec{\tau} \approx \vec{0}$), $T(\vec{0}) = \max_i D_i\beta$. For classical bandwidth-optimal algorithms (e.g., Ring), $D_i = q$ for all i (Corollary 2.2), whereas for asymmetry-resilient schedules $\max_i D_i = D_1 > q$, following directly from Theorem 2.1 and the fact that $D_{i-1} \geq D_i$ for all i. Therefore, under perfect temporal symmetry our model recovers the classical verdict: bandwidth-optimal algorithms are unbeatable, and asymmetry resilience comes with a cost.

Key Insight. Our cost model (Eq. 3) prescribes what makes a collective algorithm natively resilient to temporal asymmetry. An asymmetry-resilient algorithm's data transfer schedule should produce D_i values (i.e., the critical-path bytes that depend on rank i's arrival) that are "spread out," ideally, with smaller D_i for later arrivals to constrain their influence on the schedule. Bandwidth-optimal algorithms have a flat profile of $D_i = q$ for all i, which makes them least resilient to temporal asymmetry. The two-phase split algorithm from recent work [14], shown in Figure 1b, is the simplest example of achieving a spread in D_i values with one rank. We posit that between these two lies a rich space of collective algorithms that spread D_i across many arrivals, each tailored to a different arrival process while minimizing β cost.

Evidence from recent work. Recent work proposes an AllReduce algorithm that achieves $q_{\text{res}} = \frac{n+\log n-2}{n-1}S$, which is $\sim 2\times$ better than the bandwidth optimal q (§2), by exploiting the final arrival gap ($\tau_n - \tau_{n-1}$) [14]. For AllGather, a loose lower bound is $q_{\text{res}} \geq \frac{S}{n}$, motivating up to an $(n-1)\times$ improvement over the classical lower bound of $q = \frac{n-1}{n}S$ [46].

New algorithmic design space. Our revised model reframes algorithm design as optimizing Eq. 3 by shaping rank dependence profiles D_i against a stochastic and partially observable arrival process. The challenge is to balance the

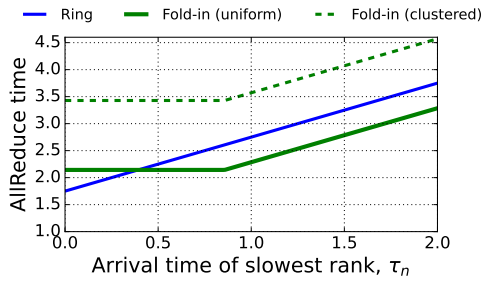


Figure 5: AllReduce under our model (n=8, β only). With uniform arrivals, each fold-in can be hidden in the gap before the next arrival (Eq. 4) while clustered arrivals (here, last 2 ranks arriving together at τ_n) serialize on the critical path.

algorithm's choice of D_i with efficient communication that transmits few (if any) redundant bytes, in order to ensure that asymmetry resilience provides speedups over bandwidth optimality. As recent work shows [14], even algorithms that solely exploit the final arrival gap require novel design. They cannot naively broadcast the final rank's data upon arrival because in most collectives, the slowest rank must both send *and* receive data that it is missing from other ranks. Moreover, these algorithms are carefully designed to minimize the precondition work so that it can be realistically overlapped with the slowest rank's delay to yield benefits in practice.

While recent work provides an algorithm for the final $\tau_n - \tau_{n-1}$ gap, we consider the general problem of asymmetry-resilient algorithm design with stochastic arrivals. Now, we must consider both the observed arrival process ($\vec{\tau}$) and the optimal set of ranks to include in every prefix of the schedule (D_i). For example, consider a *fold-in* AllReduce algorithm that treats each arrival as the straggler of the current set of "ready" ranks. This composes many executions of a recently proposed algorithm [14], each triggered by the i-th arrival.

However, each fold-in must complete before the next arrival to avoid serializing later executions. If rank i+1 arrives before fold-in i completes, its fold-in cannot begin until the previous transfers finish, causing these residuals to pile up on the critical path. Ironically, this serialization can make some asymmetry-resilient algorithms underperform the classical regime (i.e., waiting and using a bandwidth-optimal algorithm), even with temporal asymmetry. While large $\tau_i - \tau_{i-1}$ gaps enable the fold-in algorithm to perform well (Figure 4b), clustered arrivals serialize consecutive fold-ins on the critical path (Figure 4c). Ultimately, the *shape* of the arrival process is key, as Figure 5 simulates. Optimal algorithms with Eq. 3 have a dependency structure that minimizes communication cost given some arrival process $\vec{\tau}$. We note that on non-uniform topologies, the $D_i\beta$ term in Eq. 3 must weight each byte by the β of the link it traverses. While our cost model focuses on designing bandwidth-efficient algorithms, we leave augmenting it with latency terms (α) for future work.

4 Research Agenda

We identify three open questions on the path to practical asymmetry-resilient collectives.

Q1: Which collectives are amenable to asymmetry resilience, and what are the lower bounds?

While we have used AllReduce as a motivating example, distributed ML relies on many other collectives (§2). To identify which collectives may benefit most from asymmetry resilience, we must quantify the potential gains and derive new lower bounds for various collectives. For example, our analysis in §3 shows AllGather could gain nearly an $n\times$ speedup in residual communication over today's bandwidth-optimal bound, making it a natural next target. This analysis will also rule out settings where the minimum achievable $q_{\text{res}} = q$, meaning that no gain is possible.

Benefits of asymmetry-resilient algorithms also depend on the arrival process of ranks. When arrivals are well spaced out, composing prior work [14] into a fold-in algorithm (§3) achieves optimal AllReduce, whereas clustered arrivals cause it to underperform (Figure 5). Synthesizing algorithms that navigate this complex stochastic design space presents a challenging combinatorial optimization problem.

Q2: How do we decide which algorithm to execute at runtime based on the stochastic arrival process?

Asymmetry-resilient algorithms exhibit variable performance depending on the arrival process. We can either design many collective algorithms, each tailored to a $\vec{\tau}$ regime, and execute them dynamically at runtime based on the predicted arrival process, or we could seek out a small set of algorithms that are *oblivious* to the exact arrival process but perform well *in expectation*. The former is an optimal stopping problem that requires deciding whether to execute an algorithm based on the observed arrival process thus far at runtime. The latter requires understanding empirical rank arrival distributions in ML workloads and designing "universal" algorithms that minimize the *expected* communication cost. However, both change our static view of collectives into one that incorporates the stochastic nature of rank arrivals. While some algorithms may only be effective when executed under specific arrival distributions, requiring either a runtime monitor of rank arrivals at the collective or settings with persistent/pre-profiled stragglers (e.g., heterogeneous architectures, consistent load imbalance, degraded hardware), others may perform well even under perfect temporal *symmetry* (i.e., simultaneous arrivals captured by the classical regime). For example, recent work's asymmetry-resilient AllReduce algorithm is near-optimal in bandwidth costs even when all ranks arrive simultaneously, requiring minimal assumptions about rank arrival times [14].

Q3: How do we dynamically execute asymmetry-resilient algorithms in collective libraries?

Libraries like NCCL implement a collective algorithm as a schedule of data transfers. As a result, the schedule must designate the slowest rank's identity at compile time, while the actual slowest rank is only revealed at runtime. If ranks are not equally likely to be the slowest, one could compile the schedule with fixed rank identities based on prior profiling. However, this approach is suboptimal when the compile-time assumption differs from the runtime state and may generalize poorly.

NCCL already selects between algorithms (e.g., Ring vs. Tree) at runtime based on the message size [20]. We propose extending this by implementing a lightweight control plane that allows ranks to signal readiness to a coordinator. Based on the current set of rank arrivals, the control plane can leverage the online techniques from Q2 to decide which algorithm to execute. Using this decision, it can either select and execute a pre-compiled schedule for the current state or use *just-in-time (JIT) compilation* to generate the optimal schedule at runtime.

5 Related Work

Collective algorithms. Classical collective algorithms [41] achieve bandwidth or latency lower bounds under temporal symmetry on homogeneous networks and are widely used for distributed ML [20]. Recent works synthesize algorithms for heterogeneous topologies [4, 5, 9, 18, 25, 28, 33, 34, 44, 48, 54, 58]. Other works target specific topologies like tori [10, 21, 24, 47] and reconfigurable networks [26, 50, 59], or rely on existing algorithms while modifying the transport [53]. However, these algorithms all assume temporal symmetry and therefore at best achieve classical bounds that can be circumvented under temporal asymmetry. While recent work exploits the final arrival gap to speed up AllReduce [14], it cannot exploit delays for earlier arrivals and is one point in the broader design space we characterize. Theorem 2.1 highlights a fundamental tension between asymmetry resilience and classical bandwidth optimality. Thus, our goal is not only to design efficient algorithms under a revised cost model, but also to reason about the challenging online problem of when to execute them at runtime. Asynchronous and partial collectives [29] relax collective semantics and correctness (e.g., substituting stale or zero contributions for late ranks), whereas asymmetry-resilient algorithms preserve exact semantics (i.e., ensuring correctness) and instead restructure the schedule's data dependencies.

PAP-aware MPI. HPC researchers have explored how process arrival patterns (PAP) impact MPI collectives [2, 3, 15, 35, 43], including PAP-aware Reduce [35] and AllReduce [2]. These are point solutions for specific collectives that either require knowing arrival times in advance [35] or reorder existing algorithms with systems optimizations [2]. They do not establish lower bounds, chase bandwidth optimality, or describe when asymmetry is exploitable. We instead provide a framework for algorithm design under temporal asymmetry, grounded in the principle of residual communication.

Compute-communication overlap. Recent works enable compute-communication overlap for distributed ML [6, 8, 22, 30, 32, 34, 42, 49, 52]. However, they primarily trace dependencies to identify overlap opportunities at the *operator* level, rather than the design space of intra-algorithm overlap.

Communication stragglers. Several recent works propose schemes to isolate and debug communication stragglers [7, 12, 56, 57], which induce asymmetry *within* data transfers during the collective. In contrast, we propose designing new collective schedules by leveraging temporal asymmetry *before* the collective, typically arising from variation in preceding computation times across ranks.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Stefan Schmid, for their thoughtful feedback. This work was supported in part by the National Science Foundation under Awards 2444537 and 2435852. It was also supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. AD is supported by the NSF Graduate Research Fellowship.

References

[1] Osayamen Aimuyo, Byungsoo Oh, and Rachee Singh. 2025. FlashMoE: Fast Distributed MoE in a Single Kernel. *Advances in Neural Information Processing Systems* 38 (2025).

[2] Pedram Alizadeh, Amirhossein Sojoodi, Yiltan Hassan Temucin, and Ahmad Afsahi. 2022. Efficient process arrival pattern aware collective communication for deep learning. In *Proceedings of the 29th European MPI Users' Group Meeting*. 68–78.

[3] Majid Salimi Beni, Biagio Cosenza, and Sascha Hunold. 2024. MPI collective algorithm selection in the presence of process arrival patterns. In *2024 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 108–119.

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

[5] Jiamin Cao, Shangfeng Shi, Jiaqi Gao, Weisen Liu, Yifan Yang, Yichi Xu, Zhilong Zheng, Yu Guan, Kun Qian, Ying Liu, et al. 2025. SyCCL: Exploiting symmetry for efficient collective communication scheduling. In *Proceedings of the ACM SIGCOMM 2025 Conference*. 645–662.

[6] Chang Chen, Xiuhong Li, Qianchao Zhu, Jiangfei Duan, Peng Sun, Xingcheng Zhang, and Chao Yang. 2024. Centauri: Enabling efficient scheduling for communication-computation overlap in large model training via communication partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 178–191.

[7] Peiqing Chen, Jiedong Jiang, Nengneng Yu, Yuefeng Wang, Sixian Xiong, Wei Wang, and Zaoxing Liu. 2026. Don't Let a Few Network Failures Slow the Entire AllReduce. *arXiv preprint arXiv:2606.01680* (2026).

[8] Shenggan Cheng, Shengjie Lin, Lansong Diao, Hao Wu, Siyu Wang, Chang Si, Ziming Liu, Xuanlei Zhao, Jiangsu Du, Wei Lin, et al. 2025. Concerto: Automatic Communication Optimization and Scheduling for Large-Scale Deep Learning. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 198–213.

[9] Meghan Cowan, Saeed Maleki, Madanlal Musuvathi, Olli Saarikivi, and Yifan Xiong. 2023. MSCCLang: Microsoft collective communication language. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 502–514.

[10] Daniele De Sensi, Tommaso Bonato, David Saam, and Torsten Hoefler. 2024. Swing: Short-cutting rings for higher bandwidth allreduce. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 1445–1462.

[11] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56 (2013), 74–80. http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext

[12] Yangtao Deng, Lei Zhang, Qinlong Wang, Xiaoyun Zhi, Xinlei Zhang, Zhuo Jiang, Haohan Xu, Lei Wang, Zuquan Song, Gaohong Liu, et al. 2025. Mycroft: Tracing Dependencies in Collective Communication Towards Reliable LLM Training. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. 254–269.

[13] Arjun Devraj, Lindsey Bowen, and Rachee Singh. 2026. Understanding the Synchronization Tax in GPU Scale-Up Domains. *arXiv preprint arXiv:2608.22503* (2026).

[14] Arjun Devraj, Eric Ding, Abhishek Vijaya Kumar, Robert Kleinberg, and Rachee Singh. 2025. Efficient AllReduce with Stragglers. *arXiv preprint arXiv:2505.23523* (2025).

[15] Ahmad Faraj, Pitch Patarasuk, and Xin Yuan. 2007. A study of process arrival patterns for MPI collective operations. In *Proceedings of the 21st annual international conference on Supercomputing*. 168–179.

[16] Alicia Golden, Michael Kuchnik, Samuel Hsia, Zachary DeVito, Gu-Yeon Wei, David Brooks, and Carole-Jean Wu. 2025. PRISM: Probabilistic Runtime Insights and Scalable Performance Modeling for Large-Scale Distributed Training. *arXiv preprint arXiv:2510.15596* (2025).

[17] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[18] Chenyang Hei, Fuliang Li, Jiayi Li, Jiamin Cao, Chengxi Gao, Xiuzhu Sha, Tongrui Liu, Dengke Zhang, Ennan Zhai, and Xingwei Wang. 2026. HeteCCL: Synthesizing Near-Optimal Collective Communication Schedules for Heterogeneous GPU Clusters. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. USENIX Association, Renton, WA, 2533–2551. https://www.usenix.org/conference/nsdi26/presentation/hei

[19] Roger W. Hockney. 1994. The communication challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Comput.* 20, 3 (1994), 389–398.

[20] Zhiyi Hu, Siyuan Shen, Tommaso Bonato, Sylvain Jeaugey, Cedell Alexander, Eric Spada, James Dinan, Jeff Hammond, and Torsten Hoefler. 2025. Demystifying NCCL: An in-depth analysis of GPU communication protocols and algorithms. In *2025 IEEE Symposium on High-Performance Interconnects (HOTI)*. IEEE, 48–59.

[21] Nikhil Jain and Yogish Sabharwal. 2010. Optimal bucket algorithms for large MPI collectives on torus interconnects. In *Proceedings of the 24th ACM International Conference on Supercomputing*. 27–36.

[22] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. 2022. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 402–416.

[23] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. 2024. MegaScale: Scaling large language model training to more than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 745–760.

[24] Anton Juerss, Vamsi Addanki, and Stefan Schmid. 2026. Trivance: Latency-Optimal AllReduce by Shortcutting Multiport Networks. In *Proceedings of the ACM SIGCOMM 2026 Conference*. 160–176.

[25] Heehoon Kim, Junyeol Ryu, and Jaejin Lee. 2024. TCCL: Discovering better communication paths for PCIe GPU clusters. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 999–1015.

[26] Abhishek Vijaya Kumar, Arjun Devraj, and Rachee Singh. 2025. PCCL: Photonic circuit-switched collective communication for distributed ML. arXiv:2509.15450 [cs.DC] https://arxiv.org/abs/2509.15450

[27] Marco Kurzynski, Shaizeen Aga, and Di Wu. 2025. Lit Silicon: A Case Where Thermal Imbalance Couples Concurrent Execution in Multiple GPUs. *arXiv preprint arXiv:2511.09861* (2025).

[28] Yiran Lei, Dongjoo Lee, Liangyu Zhao, Daniar Kurniawan, Chanmyeong Kim, Heetaek Jeong, Changsu Kim, Hyeonseong Choi, Liangcheng Yu, Arvind Krishnamurthy, et al. 2026. FAST: An Efficient Scheduler for All-to-All GPU Communication. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. 2515–2531.

[29] Shigang Li, Tal Ben-Nun, Salvatore Di Girolamo, Dan Alistarh, and Torsten Hoefler. 2020. Taming unbalanced training workloads in deep learning with partial collective operations. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 45–61.

[30] Wanchao Liang, Tianyu Liu, Less Wright, Will Constable, Andrew Gu, Chien-Chin Huang, Iris Zhang, Wei Feng, Howard Huang, Junjie Wang, et al. 2025. TorchTitan: One-stop PyTorch native solution for production ready LLM pretraining. In *International Conference on Learning Representations*, Vol. 2025. 92161–92181.

[31] Jinkun Lin, Ziheng Jiang, Zuquan Song, Sida Zhao, Menghan Yu, Zhanghan Wang, Chenyuan Wang, Zuocheng Shi, Xiang Shi, Wei Jia, et al. 2025. Understanding stragglers in large model training using what-if analysis. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. 483–498.

[32] Tongrui Liu, Chenyang Hei, Fuliang Li, Chengxi Gao, Jiamin Cao, Tianshu Wang, Ennan Zhai, and Xingwei Wang. 2025. ResCCL: Resource-efficient scheduling for collective communication. In *Proceedings of the ACM SIGCOMM 2025 Conference*. 55–70.

[33] Xuting Liu, Behnaz Arzani, Siva Kesava Reddy Kakarla, Liangyu Zhao, Vincent Liu, Miguel Castro, Srikanth Kandula, and Luke Marshall. 2024. Rethinking machine learning collective communication as a multi-commodity flow problem. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 16–37.

[34] Kshiteej Mahajan, Ching-Hsiang Chu, Srinivas Sridharan, and Aditya Akella. 2023. Better together: Jointly optimizing ML collective scheduling and execution planning using SYNDICATE. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 809–824.

[35] Petar Marendic, Jan Lemeire, Dean Vucinic, and Peter Schelkens. 2016. A novel MPI reduction algorithm resilient to imbalances in process arrival times. *The Journal of Supercomputing* 72, 5 (2016), 1973–2013.

[36] Message Passing Interface Forum. 2012. *MPI: A Message-Passing Interface Standard, Version 3.0.* Technical Report. https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

[37] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. 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*. 1–15.

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

[39] NVIDIA. 2025. NVIDIA NVLink and NVSwitch. https://www.nvidia.com/en-us/data-center/nvlink/. Accessed: 2025-05-10.

[40] NVIDIA. 2025. *NVIDIA Unveils Rubin CPX: A New Class of GPU Designed for Massive-Context Inference.* https://nvidianews.nvidia.com/news/nvidia-unveils-rubin-cpx-a-new-class-of-gpu-designed-for-massive-context-inference

[41] Pitch Patarasuk and Xin Yuan. 2009. Bandwidth optimal all-reduce algorithms for clusters of workstations. *J. Parallel and Distrib. Comput.* 69, 2 (2009), 117–124.

[42] Suchita Pati, Shaizeen Aga, Mahzabeen Islam, Nuwan Jayasena, and Matthew D Sinclair. 2024. T3: Transparent tracking & triggering for fine-grained overlap of compute & collectives. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 1146–1164.

[43] Jerzy Proficz and Krzysztof M Ocetkiewicz. 2021. Improving Clairvoyant: reduction algorithm resilient to imbalanced process arrival patterns. *The Journal of Supercomputing* 77, 6 (2021), 6145–6177.

[44] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. 2023. TACCL: Guiding collective algorithm synthesis using communication sketches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 593–612.

[45] Prasoon Sinha, Akhil Guliani, Rutwik Jain, Brandon Tran, Matthew D Sinclair, and Shivaram Venkataraman. 2022. Not all GPUs are created equal: characterizing variability in large-scale, accelerator-rich systems. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

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

[47] Abhishek Vijaya Kumar, Eric Ding, Arjun Devraj, Darius Bunandar, and Rachee Singh. 2026. Reconfigurable Torus Fabrics for Multi-tenant ML. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (USA) *(ASPLOS '26)*. Association for Computing Machinery, New York, NY, USA, 1547–1565. https://doi.org/10.1145/3779212.3790238

[48] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica. 2020. Blink: Fast and generic collectives for distributed ML. *Proceedings of Machine Learning and Systems* 2 (2020), 172–186.

[49] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, et al. 2022. Overlap communication with dependent computation via decomposition in large deep learning models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 93–106.

[50] Weiyang Wang, Moein Khazraee, Zhizhen Zhong, Manya Ghobadi, Zhihao Jia, Dheevatsa Mudigere, Ying Zhang, and Anthony Kewitsch. 2023. TopoOpt: Co-optimizing network topology and parallelization strategy for distributed training jobs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 739–767.

[51] Xizheng Wang, Qingxu Li, Yichi Xu, Gang Lu, Dan Li, Li Chen, Heyang Zhou, Linkang Zheng, Sen Zhang, Yikai Zhu, et al. 2025. 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)*. 541–558.

[52] Yuke Wang, Boyuan Feng, Zheng Wang, Tong Geng, Kevin Barker, Ang Li, and Yufei Ding. 2023. MGG: Accelerating graph neural networks with Fine-Grained Intra-Kernel Communication-Computation pipelining on Multi-GPU platforms. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 779–795.

[53] Ertza Warraich, Omer Shabtai, Khalid Manaa, Shay Vargaftik, Yonatan Piasetzky, Matty Kadosh, Lalith Suresh, and Muhammad Shahbaz. 2025. OptiReduce: Resilient and Tail-Optimal AllReduce for Distributed Deep Learning in the Cloud. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 685–703.

[54] William Won, Midhilesh Elavazhagan, Sudarshan Srinivasan, Swati Gupta, and Tushar Krishna. 2024. TACOS: Topology-aware collective algorithm synthesizer for distributed machine learning. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 856–870.

[55] William Won, Taekyung Heo, Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. 2023. Astra-sim2.0: Modeling hierarchical networks and disaggregated systems for large-model training at scale. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 283–294.

[56] Tianyuan Wu, Wei Wang, Yinghao Yu, Siran Yang, Wenchao Wu, Qinkai Duan, Guodong Yang, Jiamang Wang, Lin Qu, and Liping Zhang. 2025. GREYHOUND: Hunting Fail-Slows in Hybrid-Parallel Training at Scale. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. 731–747.

[57] Zhiyi Yao, Pengbo Hu, Congcong Miao, Xuya Jia, Zuning Liang, Yuedong Xu, Chunzhi He, Hao Lu, Mingzhuo Chen, Xiang Li, et al. 2025. Holmes: Localizing irregularities in LLM training with mega-scale GPU clusters. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 523–540.

[58] Liangyu Zhao, Saeed Maleki, Yuanhong Wang, Zezhou Wang, Ziyue Yang, Hossein Pourreza, and Arvind Krishnamurthy. 2026. ForestColl: Throughput-Optimal Collective Communications on Heterogeneous Network Fabrics. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. 2067–2093.

[59] Liangyu Zhao, Siddharth Pal, Tapan Chugh, Weiyang Wang, Jason Fantl, Prithwish Basu, Joud Khoury, and Arvind Krishnamurthy. 2025. Efficient Direct-Connect Topologies for Collective Communications. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 705–737.

[60] Yazhou Zu, Alireza Ghaffarkhah, Hoang-Vu Dang, Brian Towles, Steven Hand, Safeen Huda, Adekunle Bello, Alexander Kolbasov, Arash Rezaei, Dayou Du, et al. 2024. Resiliency at scale: Managing Google's TPUv4 machine learning supercomputer. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 761–774.