USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

FAST: An Efficient Scheduler for All-to-All GPU Communication

Yiran Lei, *Carnegie Mellon University and MangoBoost;* Dongjoo Lee, *MangoBoost;* Liangyu Zhao, *University of Washington;* Daniar Kurniawan, Chanmyeong Kim, Heetaek Jeong, Changsu Kim, and Hyeonseong Choi, *MangoBoost;* Liangcheng Yu, *University of Pennsylvania;* Arvind Krishnamurthy, *University of Washington;* Justine Sherry, *Carnegie Mellon University;* Eriko Nurvitadhi, *MangoBoost*

https://www.usenix.org/conference/nsdi26/presentation/lei-yiran

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

FAST: An Efficient Scheduler for All-to-All GPU Communication

Yiran Lei *Dongjoo Lee* *Liangyu Zhao* *Daniar Kurniawan*
Chanmyeong Kim *Heetaek Jeong* *Changsu Kim* *Hyeonseong Choi*
Liangcheng Yu *Arvind Krishnamurthy* *Justine Sherry* *Eriko Nurvitadhi*

Carnegie Mellon University *MangoBoost* *University of Washington* *University of Pennsylvania*

Abstract

All-to-All(v) communication is a critical primitive in modern machine learning workloads, particularly mixture-of-experts (MoE) models. Unfortunately, efficient scheduling is challenging due to workload skew, heterogeneous two-tier fabrics, and incast congestion, compounded by the dynamic nature of MoE workloads, where traffic shifts every few hundred milliseconds. Existing schedulers are hardly scalable, incurring seconds to hours of synthesis time, making them impractical.

We present FAST, an efficient All-to-All(v) scheduler. FAST addresses skew through intra-server rebalancing and enforces balanced, one-to-one scale-out transfers that avoid incast. Evaluated extensively on both NVIDIA H200 and AMD MI300X clusters, FAST consistently outperforms state-of-the-art solutions on skewed workloads while reducing synthesis time by orders of magnitude.

1 Introduction

All-to-All communication—where every endpoint sends data to all others—has long been a fundamental collective communication primitive in scientific and parallel computing, supporting workloads such as 3D FFTs [48]. Early systems typically treated each server as the communication endpoint, interconnected by networks with relatively uniform bandwidth. In modern ML clusters, the endpoint has shifted to individual GPUs, which are connected through a two-tier fabric consisting of faster intra-server (scale-up) links and slower inter-server (scale-out) links.[1] Consequently, All-to-All has become a key operation for many ML applications, including recommendation models [35, 36], Gaussian Splatting [56], and mixture-of-experts (MoE) models [15, 22, 27, 50].

The role of All-to-All in ML applications is important. In mixture-of-experts (MoE) models in particular, its cost can account for a large fraction of training time. MoE improves the efficiency of model parameters by activating only a subset of experts for each input token rather than all simultaneously. This selectivity, however, necessitates frequent All-to-All operations to dispatch tokens to experts across GPUs and aggregate their outputs. Prior studies [22, 29] show that MoE All-to-All can consume 30–55% of training time, making it a major contributor to overhead in large-scale training.

The challenges of All-to-All arise at both the application and system layers. At the application layer, traffic is often **skewed** and **dynamic**. In MoE, some experts are selected more frequently than others, leading to larger data transfers for their corresponding GPUs during All-to-All. This imbalance keeps certain GPUs and NICs busy after others have finished, creating straggler effects. When communication volumes differ across endpoints in All-to-All, the operation is referred to as `alltoallv` [34]. Further, the traffic pattern in `alltoallv` changes every few hundred milliseconds, as the MoE gating function reassigns tokens to experts at runtime (Figure 1). As a result, a GPU that is a hotspot at one moment may be idle the next. This dynamism makes static schedules impractical and requires schedulers to adapt in real time.

At the system layer, the hardware fabric that connects GPUs is inherently two-tiered: fast intra-server links (scale-up) and much slower inter-server links (scale-out) (Figure 4). This **heterogeneity** means that flows of the same size can finish quickly inside a server but take an order of magnitude longer across servers, leaving schedulers to coordinate thousands of flows over mismatched bandwidths. In addition, `alltoallv`'s dense communication pattern naturally triggers **incast**, a classic networking problem where many senders overload downlink of the same receiver. Incast causes network congestion with switch queue buildup and reduced goodput—even under modern congestion control—and remains an open challenge in large-scale cluster networking [13, 21].

Together, these challenges make efficient `alltoallv` scheduling especially difficult under the tight timescales demanded by MoE. Existing schedulers such as TACCL [53] and TE-CCL [31] employ solvers [20] to generate near-optimal schedules. While they overcome the inefficiencies of fixed schedules in collective communication libraries such as NCCL and RCCL, their resulting formulations are NP-

[1] The terms 'scale-up' and 'intra-server' network are used interchangeably, and likewise 'scale-out' and 'inter-server'.

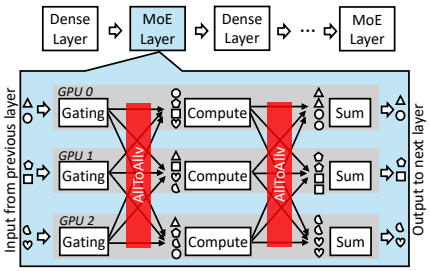
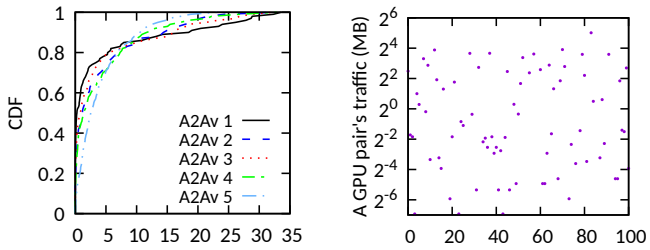


Figure 1: MoE models invoke `alltoallv` twice per MoE layer, making it a critical communication primitive.



(a) Skewness (b) Dynamism

Figure 2: All-to-All workload is skewed and dynamic when using Megatron-LM to pre-train a MoE model.

hard [53] and can take minutes to hours to synthesize a schedule even for just 32 GPUs. The state-of-the-art scheduler SyCCL [11] accelerates this process with heuristics and parallelism, yet requires seconds to minutes for balanced All-to-All, while skewed `alltoallv` still remains unresolved. Consequently, these schedulers are far too slow for MoE `alltoallv` workloads that shift every few hundred milliseconds.

In this paper, rather than increasing scheduler complexity, we simplify the problem itself: *It suffices to focus on optimizing the scale-out tier—the real bottleneck*. We observe that scale-up is roughly an order-of-magnitude faster than scale-out (Figure 4b). So the much faster scale-up fabric can cheaply absorb skew within each server, reshaping traffic before it reaches scale-out. With this, we can then maximize scale-out efficiency by keeping bottleneck servers transmitting at full rate while avoiding incast. Achieving this requires pairing senders and receivers without contention, which reduces naturally to a one-to-one matching problem between endpoints, solvable in polynomial time.

Building on this insight, we propose FAST, a polynomial-time, matching-based scheduler for skewed and dynamic `alltoallv` workloads. FAST operates in two phases: (i) *Skew mitigation*, where it uses fast scale-up fabric to rebalance the scale-out workload so that all NICs face equal volume before traversing the slow scale-out links; and (ii) *Balanced, one-to-one transfers*, which use Birkhoff's decomposition [9] to generate successive matchings between senders and receivers, ensuring scale-out transfers proceed without incast while keeping bottleneck servers fully active at line rate until completion. While Birkhoff's decomposition has appeared in the design of switches [12, 30, 52], to our knowledge, this is the first work to apply Birkhoff's decomposition to scheduling collective communication at GPU endpoints.

We implement FAST on both NVIDIA H200 [44] and AMD MI300X [2] testbeds and evaluate it against state-of-the-art solutions such as DeepEP [55], TACCL [53], and TE-CCL [31]. Under skewed workloads, FAST outperforms the strongest NVIDIA baseline by 1.01–1.3× and the strongest AMD baselines by 1.5–2.8×, and when integrated into Megatron-LM [54] on AMD, improves end-to-end MoE training throughput by 4.48× over RCCL [6] that suffers heavily from incast. The scheduler is highly efficient, completing in 221 µs for 64 GPUs—fast enough for MoE workloads where the traffic matrix changes every few hundred milliseconds. Our code is available at [26].

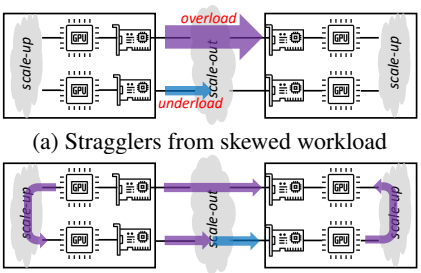
2 Motivation

All-to-All (`alltoallv`) communication—where every endpoint exchanges distinct data with all others—has become a key primitive in modern GPU clusters. In this setting, the communication endpoint is each individual GPU, connected by a two-tier fabric consisting of fast intra-server (scale-up) links and slower inter-server (scale-out) links.

All-to-All communication cost in MoE models. Mixture-of-experts (MoE) is now a leading architecture for scalable large language models: instead of activating the full model for every input token, a lightweight gating network selects only a subset of 'experts', each implemented as a feed-forward network (FFN). While expert parallelism (EP) improves parameter efficiency, it introduces frequent, large-scale `alltoallv` operations to dispatch tokens to selected experts—often spanning hundreds of GPUs [7, 15]—and to gather results. As illustrated in Figure 1, each MoE layer invokes `alltoallv` twice, and MoE layers often constitute a large fraction of the model. Prior measurements [22, 29] have shown that `alltoallv` can account for 30–56% of training time.

While MoE is our primary focus, the importance of `alltoallv` extends beyond. It underpins recommendation systems [35, 36], Gaussian Splatting [56], and classical scientific workloads such as 3D FFT [48] (where it can dominate up to 97.3% of runtime [8]!). Despite being just one collective, `alltoallv` performance disproportionately shapes the efficiency of both modern AI and traditional HPC workloads.

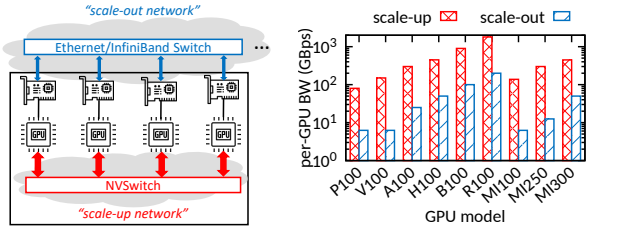
Application challenges—skewness & dynamism. To study this communication, we profile MoE training using Megatron-LM [54] with 32 experts (one per GPU). We find that `alltoallv` workloads are inherently *skewed* and *dynamic*, consistent with recent profiling results of Mixtral models [23, 29]. Unlike balanced collectives such as All-Reduce, MoE `alltoallv` generates a highly uneven demand matrix: some GPU pairs exchange more than 12× the median volume (Figure 2a). This skew creates *stragglers*—NICs that remain busy long after others have finished, delaying the en-



(a) Stragglers from skewed workload

(b) Mitigation via rebalanced traffic

Figure 3: Workload skew creates stragglers and underutilized NICs that degrade performance, which can be mitigated through traffic rebalancing.



(a) Two-tier structure (b) Per-GPU full-duplex bandwidth

Figure 4: Modern GPU clusters feature a two-tier fabric: a high-bandwidth intra-server scale-up network (e.g., NVLink, Infinity Fabric) and a lower-bandwidth inter-server scale-out network (e.g., Ethernet, InfiniBand).

tire collective and stalling training progress (Figure 3a). In this setting, scheduling is critical: by routing part of a heavy GPU's traffic to idle NICs, a smart scheduler can smooth out skew and mitigate stragglers, as illustrated in Figure 3b.

MoE workloads are also *dynamic*: `alltoallv` traffic pattern changes every few hundred milliseconds. As shown in Figure 2b, the traffic volume of a given GPU pair can vary significantly across successive `alltoallv` invocations, since token routing is jointly decided by the input tokens and per-MoE-layer gating functions (Figure 1) and cannot be predicted in advance. Therefore, communication schedules must be recomputed online at the timescale as workload changes—a GPU pair with heavy load in one `alltoallv` may be nearly idle in the next—making fast, online scheduling essential.

System challenges—heterogeneity & incast. Modern GPU clusters further complicate scheduling with two system-level challenges. First, *a heterogeneous, two-tier fabric* connects GPUs (Figure 4a): fast intra-server links (e.g., 5th-gen NVLink 900 GBps [43]) and much slower inter-server links (e.g., Ethernet 800 Gbps [42]). This two-tier network increases scheduling complexity: for each `alltoallv`, schedulers must navigate thousands of flows across mismatched bandwidths, exploring a large space of routing and pacing decisions, which can turn scheduling itself into a bottleneck.

Second, *incast* is a classic networking problem that arises from `alltoallv`'s dense communication pattern, where many flows converge on the same NIC downlink in the scale-out fabric. While the burstiness of small messages can be absorbed by switch queues, MoE `alltoallv` transfers are much larger—typically 100 MB to 1 GB [55]—causing sustained congestion that requires active control. Even with advanced schemes [19, 21, 33], incast often results in unfair bandwidth sharing and degraded goodput. It remains an open challenge [13], and schedulers often attempt to mitigate it proactively rather than relying on the transport layer alone.

Limitations of existing approaches. State-of-the-art schedulers such as TACCL [53], TE-CCL [31], and SyCCL [11] are designed to be *general*: they support a wide range of collectives (All-Reduce, All-Gather, All-to-All, etc.) and reason about arbitrary topologies. To achieve this generality, they cast scheduling as constraint-satisfaction or optimization problems, often NP-hard [53], and solve them with heavy computation. This is practical for collectives with *repetitive* communication patterns like All-Reduce, where the high scheduling cost can be amortized over many iterations.

For `alltoallv`, however, existing approaches are too slow. SyCCL [11], the fastest to date with parallelism and heuristic acceleration, is orders of magnitude faster than earlier solver-based systems but still requires minutes to produce a schedule for 64 GPUs—impractical when MoE traffic shifts every few hundred milliseconds. While such systems excel at achieving near-optimal completion under arbitrary topologies, their fine-grained modeling makes them hardly scalable to this setting.

At the other extreme, production libraries like NCCL [4] generate schedules instantly, but rely on fixed schedules oblivious to the dynamic, skewed workload, often resulting in lower throughput than what the hardware could achieve.

Goal. Can we design a fast, online scheduler for `alltoallv` that sustains high performance? Instead of targeting arbitrary collectives or topologies, we focus on a specialized solution for `alltoallv` on today's two-tier GPU clusters—where skewness, dynamism, asymmetry, and incast constrain existing schedulers. FAST integrates with existing libraries: the runtime dispatches `alltoallv` to FAST and uses conventional algorithms for other collectives.

3 Design Overview

To generate a separate schedule for each `alltoallv` invocation in modern GPU clusters, we take a step back. Rather than enumerating complex constraints, we start by first solving `alltoallv` on a simplified single-tier network and then generalize the solution to today's asymmetric two-tier fabrics.

Starting point: `alltoallv` in a single-tier network. On a single-tier, full-bisection network with uniform link bandwidth, the objective is to maximize communication efficiency by avoiding incast and congestion. This is achieved by ensuring that, at any instant, each sender communicates with exactly one receiver and each receiver accepts data from exactly one sender. As a result, communication can be organized into stages, where each stage realizes a one-to-one matching between senders and receivers, and successive stages collec-

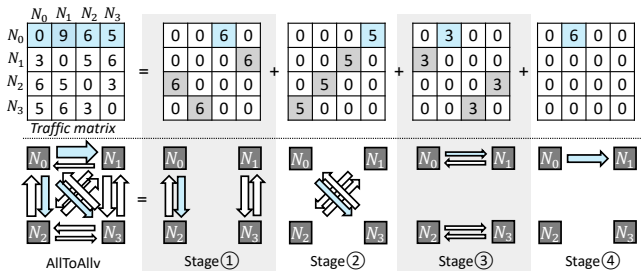


Figure 5: Birkhoff's decomposition of a 4-node `alltoallv`. Completion time is dictated by the largest sender (N_0 in blue), and the schedule is optimal since N_0 stays active in every stage while lighter nodes drop out early.

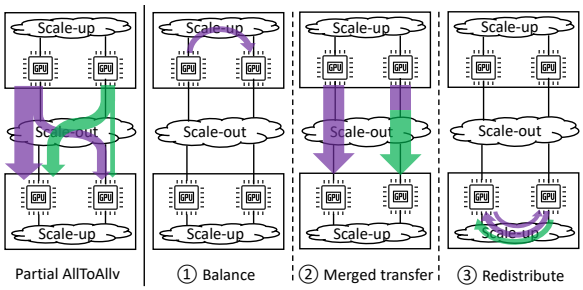


Figure 6: FAST: transforming an `alltoallv` workload with sender/receiver stragglers into a balanced scale-out transfer. Each GPU is connected to a dedicated NIC (omitted).

tively exchange data across all sender–receiver pairs.

For such a schedule to reach the theoretical minimum completion time, two conditions must hold: (i) each stage is balanced so all active nodes start and finish together, and (ii) the bottleneck endpoints (the heaviest senders or receivers) remain fully active at line rate until completion.

To solve this problem, we observe that Birkhoff's decomposition [9]—introduced in 1946 as a mathematical theorem—can be reinterpreted as an optimal scheduling strategy for `alltoallv`. Formally, the theorem states that any traffic matrix[2] can be expressed as a weighted sum of *permutation matrices*. Viewed through the scheduling lens, each permutation corresponds to a transfer stage: every active row (sender) and column (receiver) has exactly one nonzero entry of equal size (transfer size), so each participant exchanges data with exactly one partner and all finish the stage together. By summing over these permutation matrices, all flows advance in proportion to their demand, driving the transfer to completion.

Figure 5 illustrates our strawman approach: the top pane shows the decomposition into (partial) permutation matrices, while the bottom pane shows the corresponding transfers. This approach is appealing because (i) completion time hits the lower bound—the bottleneck node (e.g., N_0 as sender in blue) transmits in *all* stages; (ii) each stage is balanced until nodes finish, so participants advance to the next stage together; and (iii) the decomposition is computationally efficient.

Challenges of applying Birkhoff's decomposition to two-tier networks. Unfortunately, modern GPU clusters deviate from the simplified network setting in two ways. (i) In a two-tier fabric, even a 'balanced' permutation stage can finish unevenly—transfers on the inter-server links lag behind, idling the faster intra-server links and wasting bandwidth. (ii) In 8-GPU-per-server (e.g., HGX [46]) clusters, completion time is dictated by the busiest inter-server endpoints. Because most GPU pairs span server, the faster intra-server tier is rarely limiting. Under heavy skew, Birkhoff-based schedules cannot relieve this inter-server bottleneck and therefore still stall.

Our approach. We exploit the two-tier fabric as an opportunity to simplify scheduling. Since intra-server (scale-up) bandwidth is far higher than inter-server (scale-out), it is rarely a bottleneck. We instead repurpose it to *reshape traffic in advance*, absorbing imbalance locally so the scale-out tier sees a more uniform workload and Birkhoff's decomposition can produce more efficient schedules.

At the heart of this reshaping is another simple yet powerful observation: *what matters is delivering data to the right servers*. Which GPU inside a server handles the transfer is secondary, since intra-server shuffles are cheap compared to scale-out communication. This leads to a two-phase design:

Intra-server scheduling: balancing and redistribution (§4.1). Within each server, we equalize traffic across GPUs before it leaves the node (Figure 6): overloaded GPUs hand off excess traffic to lighter ones so every NIC carries the same volume per destination server. On the receiving side, each GPU receives data from exactly one designated sender on each source server, equalizing incoming volume. This process makes some data initially arrive at a 'proxy' GPU at the correct destination server, which is then quickly forwarded to the true destination GPU via a cheap intra-server redistribution.

Inter-server scheduling: balanced one-to-one transfers (§4.2). Once intra-server skew is absorbed, the remaining challenges are server-level imbalance and incast. Here, we apply Birkhoff's decomposition to construct successive one-to-one, balanced transfer stages, ensuring bottleneck servers remain active at line rate until their traffic is complete.

Finally, we pipeline the two phases to tighten end-to-end transfer (§4.3), and conclude by analyzing key properties (e.g., optimality and complexity) (§4.4).

4 Two Phase Scheduler

This section presents our design: mitigating skew within servers (§4.1), handling incast and imbalance across servers (§4.2), and pipelining both to improve end-to-end transfers (§4.3). We conclude with an analysis of key properties (§4.4).

4.1 Intra-server Scheduling: Balancing and Redistribution

The first challenge arises within each server: GPUs often generate or absorb uneven volumes of traffic, creating stragglers

[2]The theorem applies to scaled doubly stochastic matrices; arbitrary matrices can be adapted, as described later.

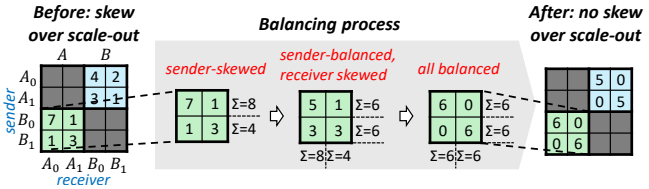


Figure 7: Balancing process for a 2-server, 2-GPU-per-server `alltoallv`: a skewed tile (left) is reshaped into a scalar form (right), ensuring no GPU NIC is overloaded. Grey tiles (intra-server) omitted for clarity.

where some NICs sit idle while others are overloaded. By leveraging the fast scale-up fabric, our goal is to eliminate these sender- and receiver-side imbalances, presenting the scale-out tier with a more uniform workload.

Example setup. Consider a simple 2-server case (A, B), each with 2 GPUs (A_0, A_1, B_0, B_1). The workload is represented by a 4×4 GPU-to-GPU traffic matrix (Figure 7). Each *row sum* reflects a GPU's total outgoing volume; each *column sum* reflects its total incoming volume. Cross-server transfers appear as 2×2 tiles (blue for $A \to B$, green for $B \to A$). Since scale-out is the bottleneck, we focus on these tiles, omitting the grey intra-server diagonals for the purpose of illustration.

Mitigating sender skew. The first step is to prevent a GPU from being the 'straggler sender'. In the $B \to A$ tile, GPU B_0 must send 8 units, while B_1 only needs 4. If transmitted directly, B_1 would finish early, leaving B_0 as a straggler. To avoid this, we rebalance within server B: heavily loaded GPUs shift part of their traffic to lightly loaded ones using the scale-up fabric. Here, B_0 transfers 2 units to B_1, so both end up with 6. In matrix terms, the *row sums* of the tile are equalized—ensuring every NIC in B contributes the same total outgoing load to server A.

Mitigating receiver skew. After sender-side balancing, GPU A_0 may still receive 8 units (column sum) while A_1 only needs 4, leaving a receiver-side straggler. The fix is to decouple the notions of 'correct server' and 'correct GPU'. Each sender forwards *all* of its traffic to its peer GPU with the same local index ($B_0 \to A_0$, $B_1 \to A_1$), ensuring data first arrives at the correct server. This *merged peer transfer* keeps receiver loads balanced—since senders were equalized earlier—even though some data arrives temporarily at the wrong GPU, to be corrected later. In matrix form, each row collapses into a single nonzero in its row-local-index column, turning the 2×2 tile into a *scalar matrix* with equal diagonal entries and all off-diagonals zero (right in Figure 7). The result is one-to-one, balanced scale-out transfers across GPUs.

Redistribution. At this point, all traffic reaches the correct *server*, but may still be at the *wrong GPU*. A final redistribution step corrects placement inside the server, routing traffic from the proxy GPU to the true destination over the scale-up fabric. Because scale-up is an order of magnitude faster than scale-out, this added step incurs small overhead.

By combining sender balancing, merged peer transfers, and

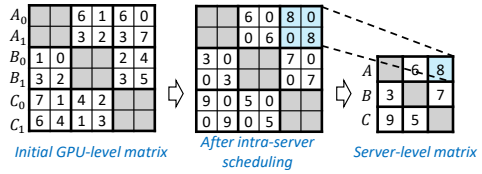


Figure 8: Intra-server scheduling reduces a 6×6 GPU-level matrix (A_0,A_1,$\cdots$,C_1) to a 3×3 server-level matrix (A,B,C).

local redistribution, our scheduler reshapes *each cross-server tile* into its most balanced *scalar form*. This transformation removes intra-server skew and equalizes scale-out traffic at GPU level. What remains is the higher-level server-to-server skew, which we address next.

4.2 Inter-server Scheduling: Balanced, One-to-one Transfers

Although intra-server scheduling reshapes the initial skewed GPU-level `alltoallv` into a more balanced form, balancing over scale-up network *within each server* cannot eliminate *server-to-server* skew. Some servers still send or receive more traffic than others, creating bottlenecks that must be addressed to achieve good end-to-end performance.

This imbalance becomes clearer once we reduce the GPU-to-GPU traffic matrix into a simpler server-to-server view. Figure 8 shows a 3-server, 2-GPU-per-server example: the original 6×6 GPU matrix (left) is reshaped by intra-server scheduling into the balanced form (middle), where each 2×2 server-to-server tile becomes a scalar matrix. Each scalar tile can then be collapsed into a single entry, yielding the reduced 3×3 server-level matrix (right). The intuition is that, after intra-server scheduling, GPUs within a server *act identically over scale-out*—each sending and receiving equal volumes—so we can abstract away individual GPUs. This reduction both exposes the remaining skew across servers and simplifies scheduling, since the server-level problem is typically an order of magnitude smaller than the GPU-level one (e.g., with 8 GPUs per server [46] in modern clusters).

At this server level, two scheduling challenges remain: (i) incast: even with peer access, GPU_j from multiple servers may funnel into GPU_j of the same destination server, overloading its scale-out link; and (ii) throughput optimality: the busiest servers must remain fully active at line rate until their traffic flows complete, otherwise the overall completion time lags behind the achievable minimum.

SpreadOut: one-to-one but not optimal. A natural incast-free baseline is MPI's SpreadOut algorithm [37], which cycles through 'shifted diagonals' of the $N \times N$ server matrix: at stage i, server s sends to server $(s+i)\%N$. This guarantees one-to-one sender-receiver mappings at every stage.

However, SpreadOut fails the second requirement: the bottleneck server may sit idle during many stages. Although the bottleneck is the row or column with the largest *sum*, the entry selected for that row or column in a given stage need not be the largest entry on the corresponding diagonal. When this

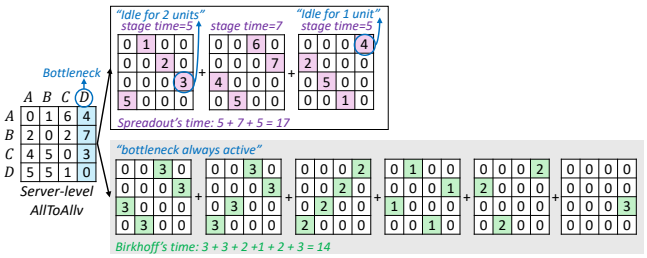


Figure 9: SpreadOut vs. Birkhoff. Both use one-to-one mappings, but SpreadOut (top) can stall at each stage by leaving the bottleneck server idle, while Birkhoff (bottom) keeps it continuously active until completion (optimal).

happens, the stage is gated by another matrix entry (i.e., a flow from non-bottlenecks), forcing the true bottleneck to wait.

Figure 9 shows an example: server D, *as a receiver*, is the bottleneck—with 14 units column sum—the heaviest among all rows/columns. But in stage 1, D receives only 3 units, while the stage is gated by a separate 5-unit flow ($D \rightarrow A$ where D is *a sender*). As a result, D sits idle *as a receiver* for 2 extra time units. The same situation recurs in stage 3, adding another 1 unit of idle time. Altogether, SpreadOut finishes in 17 units—3 units slower than the 14-unit theoretical minimum.

In matrix terms, SpreadOut's completion time equals the *sum of the maximum entry on each diagonal*. This sum is provably no smaller than the largest row or column sum—the true lower bound—so SpreadOut can not guarantee optimality.

Birkhoff's decomposition: one-to-one with optimality. The optimal completion time is determined by the busiest server—the largest row or column sum in the matrix. In Figure 9, the busiest server D must receive 14 units, so the minimum possible time is 14. Hitting this bound requires D to receive at line rate in every stage.

Birkhoff's decomposition [9] is tailored to this setting. It expresses a traffic matrix as a weighted sum of *permutation matrices*. Each permutation matrix has exactly one nonzero per row and column, all of identical value, representing a one-to-one, balanced transfer stage—every active sender transmits the same amount to exactly one receiver, and each receiver accepts from one sender. Some permutation matrices may be *partial*, with zero rows or columns for servers that have already finished. In Figure 9, the first four stages are full permutation matrices, while the last two are partial.

Viewed as a schedule, this decomposition yields a sequence of one-to-one sender-receiver matchings where bottleneck servers remain continuously active until they complete. In our example, Birkhoff finishes in 14 units—exactly the lower bound—achieving optimality.

Multi-server end-to-end scheduling. The final piece of our scheduler—Birkhoff's decomposition—addresses the remaining server-level skew. We illustrate the full scheduling process with a 3-server, 2-GPU-per-server example in Figure 10, omitting intra-server transfers (grey diagonal tiles) for clarity. The input is a 6×6 GPU-to-GPU traffic matrix (left). Without our scheduler, the completion-time lower bound is 10 units, set by the heaviest sender GPU (B_1, row sum 10) and receiver GPU (B_0, column sum 10).

Step 1: Balancing. This step reduces the severity of the bottleneck. Within each 2×2 tile (e.g., $A \rightarrow B$), sender loads are equalized across GPUs, and *peer transfer* (e.g., $A_i \rightarrow B_i$) ensures receivers share the load evenly. With intra-server skew removed, *the effective lower bound improves*: in the reshaped matrix (middle of Figure 10), the maximum row/column sum drops from 10 to 8 (i.e., A, C as sender and A, B as receiver). Intuitively, the pressure of a straggling NIC/GPU is averaged across all GPUs and NICs within that server, reducing its impact. At this point, the 6×6 GPU-level matrix can be cleanly collapsed into a skewed 3×3 server-level `alltoallv`.

Step 2: Balanced, one-to-one transfer stages. Birkhoff's decomposition then partitions this server-level matrix into three balanced, one-to-one transfer stages (right). Each stage delivers a portion of the workload, and together they complete all transfers. The resulting schedule satisfies three key properties: (i) *Incast-free:* At the server level, Birkhoff enforces one-to-one matchings. Combined with Step 1's peer-access rule, each GPU communicates *only with the same-index GPU in the matched server*, preventing any receiver overload. (ii) *Balanced:* Servers send equal volumes per stage, while Step 1 guarantees balanced GPUs within each server. (iii) *Optimal:* Bottleneck servers (i.e., A, C as senders, A, B as receiver here) remain fully active across all stages, achieving the theoretical minimum completion time (8 units).

Step 3: Per-stage redistribution. Step 2's scale-out transfers ensure data reaches the correct server, but not necessarily the correct GPU. A lightweight redistribution step fixes placement locally, aligned with each stage. For example, in Figure 10, once $A \rightarrow B$ (shown as blue tile) completes in Stage ①, the corresponding portion is immediately redistributed within B (shown as the blue-striped tile).

In summary, our scheduler completes the server-level scheduling by decomposing the reshaped workload into a sequence of balanced one-to-one stages. We now turn to how these stages are executed in practice, showing how pipelining overlaps inter- and intra-server transfers to further reduce latency and hide balancing and redistribution costs.

4.3 End-to-End Transfer Pipeline

Our scheduler combines scale-up transfers (balancing and redistribution) with scale-out transfers (staged by Birkhoff's decomposition). While scale-up is much faster than scale-out, *serializing all the steps*, e.g., `balance` → `stage 1 scale-out` → `stage 1 redistribute` → `stage 2 scale-out` → ..., still results in noticeable overhead. To minimize end-to-end completion time, we build a pipeline that keeps the scale-out tier—the true bottleneck—as busy as possible, while hiding scale-up transfers in the background.

There are three types of scale-up transfers to consider: (i) Balancing, which equalizes GPU loads before scale-out

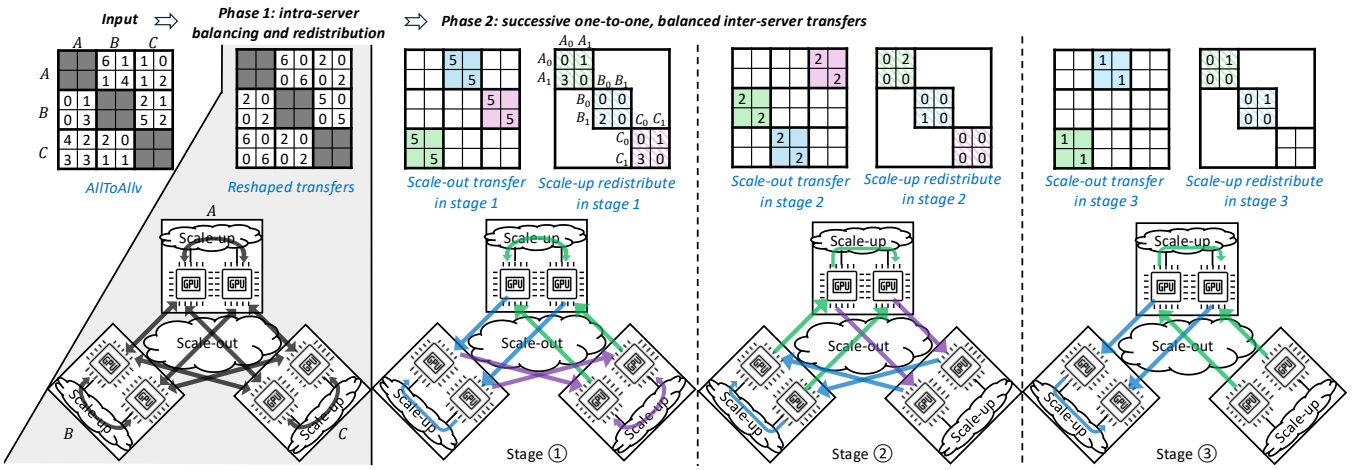


Figure 10: End-to-end scheduling example for a 3-server, 2-GPU-per-server (6×6) `alltoallv` workload. The traffic matrix (top) and transfer process (bottom) show how intra-server scheduling mitigates skew, reshaping each tile into a scalar form (left → middle). Inter-server scheduling then applies Birkhoff's decomposition to schedule successive 1-to-1 server transfers (middle → right). Each stage is balanced, 1-to-1, and keeps bottleneck servers active, achieving near-optimal scale-out performance.

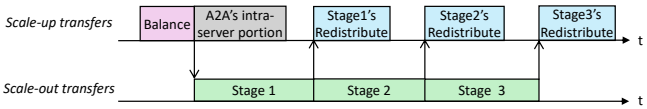


Figure 11: End-to-end pipeline: scale-out transfers are kept as active as possible while scale-up operations overlap in the background. Arrows indicate triggering between transfers.

begins; (ii) Redistribution, which corrects placement as scale-out completes, now aligned *per stage* (e.g., the striped tiles in Figure 10); and (iii) Intra-server portion of `alltoallv` (grey tiles in Figure 10). The scale-out portion is straightforward: successive one-to-one stages from Birkhoff's decomposition.

Pipeline structure. Figure 11 illustrates how the pipeline operates. Balancing (purple) runs first, as all subsequent scale-out stages depend on the reshaped workload. Once balancing finishes, scale-out begins immediately. During scale-out, Stage i's redistribution (blue) overlaps with Stage $(i+1)$'s scale-out (green), hiding redistribution cost—except in the final stage, which has no successor. The intra-server portion of `alltoallv` (grey) is executed alongside the first scale-out stage, making use of idle scale-up bandwidth before redistributions are triggered. While the pipeline could be made even tighter by subdividing balancing and scale-out into smaller chunks, the gain is small, so we adopt this simple design.

Overall, the pipeline keeps the scale-out network—the true bottleneck—as busy as possible, while scale-up operations are largely hidden in the background. This completes the design: intra-server scheduling removes GPU-level skew, inter-server scheduling produces balanced server-level transfers, and pipelining integrates them into a seamless end-to-end execution. Next, we turn to the key properties of our scheduler, including optimality and computational complexity.

4.4 Scheduler Properties

Beyond the algorithm itself, the scheduler's properties reveal why it is a practical, efficient, and high-performance solution for `alltoallv` in modern GPU clusters. Assume N servers and each server has M GPUs.

Optimality. The majority of end-to-end transfer—staged scale-out transfers generated by Birkhoff's decomposition—operates at full efficiency on the scale-out fabric. Suboptimality arises from additional intra-server operations such as balancing and redistribution. Empirically, these costs are small: under typical workloads, they add less than 5% to the total scale-out cost (as shown in Section 5.1.3), and even under highly skewed traffic (Zipfian with skewness 0.9), the overhead remains below 8%.

We also prove in Appendix A.1 that even under adversarial workloads—designed to maximize balancing (e.g., all GPUs within a server join `alltoallv` but only one holds data) and redistribution (e.g., only one GPU in each destination server receives data)—the performance gap from the theoretical optimum remains bounded. For instance, in a 4-node cluster with a 9:1 scale-up to scale-out bandwidth ratio (450 GBps 4th-gen NVLink [43] vs. 400 Gbps Ethernet), FAST finishes within 2.12× of the optimum, even in this worst-case setting.

Number of stages. The number of transfer stages produced by Birkhoff's decomposition depends on the *server-level* matrix. In the best case, a perfectly balanced $N \times N$ server-level matrix requires exactly N stages—the same as SpreadOut. With skew, more stages may be introduced, but the total is always bounded by $(N^2 - 2N + 2)$ [24].

Fewer stages are preferable because each adds synchronization overhead. Ideally, one would minimize the stage count, but finding such a decomposition is NP-hard [16]. Rather than attempting this costly search, our scheduler efficiently pro-

duces a valid decomposition. Since the stage count is bounded, the synchronization cost is also bounded—and in practice, our evaluation shows it to be negligible.

Computational complexity & runtime. Intra-server scheduling is lightweight, involving only simple load balancing and bookkeeping. The primary cost lies in inter-server scheduling—Birkhoff's decomposition—which runs in polynomial time with $O(N^5)$ complexity and is fast in practice.

As shown in Figure 16, our scheduler completes in 25 µs for 4 servers (32 GPUs), 221 µs for 8 servers (64 GPUs), and 805 µs for 12 servers (96 GPUs), assuming $M = 8$ GPUs per server—a common configuration [46]. With one expert per GPU, this spans the typical expert-parallelism (EP) range of today's MoE workloads, from EP8—EP128 (e.g., Perplexity AI [7]) to large-scale deployments such as EP320 (DeepSeek [15]). Even at EP320 (40 servers), scheduling overhead remains modest at 77 ms, well within practical bounds.

To put this in context, consider a median-scale case with EP64: each GPU transmits about 1 GB of data to others—the traffic scale reported by prior work [29, 55]. Over a 400 Gbps network, such an All-to-All takes at least 20 ms, while scheduling adds 221 µs (≈1.1% of total time). Our scheduling step is a small upfront 'tax' that yields a fully optimized plan, shortening end-to-end completion compared to no schedules.

By contrast, prior approaches [11, 31, 53] cast scheduling as NP-hard problems (e.g., MILP or multi-commodity flow) and rely on solvers [20] that run orders of magnitude slower: for a 16-GPU All-to-All, the fastest solver-based scheduler, SyCCL [11], takes 3.6 s—while our scheduler runs in 3.1 µs.

Exclusion of All-to-All scheduling over scale-up. Both balancing and redistribution are themselves skewed `alltoallv` operations. Because they run entirely on the fast scale-up fabric, sophisticated scheduling is unnecessary. For these steps we use MPI's SpreadOut algorithm [37], which provides simple one-to-one sender–receiver mappings at low cost. While Birkhoff's decomposition could deliver an optimal schedule here as well, its added computation is unnecessary—the scale-up tier is not the bottleneck.

Another caveat is that SpreadOut may not be well suited for older GPUs with non-symmetric scale-up topologies, e.g., the ring in AMD MI250 [1] and the hybrid cube mesh in NVIDIA V100 [41]. However, recent GPUs adopt symmetric scale-up fabric, e.g., switch-based topology [43] and fully connected mesh [2], which are our target platforms.

Adapting an arbitrary matrix to a valid form. Birkhoff's theorem applies to *scaled doubly stochastic matrices*, where all row and column sums are equal. Since real $N \times N$ server-level traffic matrices are arbitrary, we first embed them into this form by adding an auxiliary matrix, which can be constructed in $O(N^2)$ time. This procedure increases only the lighter rows or columns until all sums match the heaviest one, leaving the true bottleneck row or column unchanged.

The auxiliary entries represent *virtual* transfers that are never executed and are ignored once all real traffic completes. As a result, some permutation matrices produced by the decomposition may appear partial *with respect to real traffic* (Figure 9), with zero rows or columns originating from the auxiliary matrix. Importantly, this transformation preserves *both correctness and optimality, since the maximum row or column sum—the true bottleneck—remains unchanged.*

Birkhoff's decomposition: overview. The algorithm takes as input an $N \times N$ scaled doubly stochastic matrix and outputs a sequence of permutation matrices whose weighted sum reconstructs the input. The matrix can be viewed as a bipartite graph with N senders (rows) and N receivers (columns), where nonzero entries correspond to edges.

At a high level, the algorithm repeatedly finds *a perfect matching* in this graph; each such matching selects exactly one outgoing edge per sender and one incoming edge per receiver, yielding a permutation matrix. Each matching can be computed, for example, using the Hungarian algorithm [25] with $O(N^3)$ complexity. After subtracting the permutation matrix derived from each matching, the residual matrix remains scaled doubly stochastic, allowing the process to repeat. In the worst case, the algorithm requires $O(N^2 - 2N + 2)$ iterations [24], resulting in $O(N^5)$ total complexity.

A key advantage of the algorithm is that it advances *all* bottleneck rows and columns—potentially multiple with equal maximum load—at the same rate. In contract, a greedy algorithm may fail to account for all bottlenecks simultaneously, often prioritizing individual large entries and suboptimal.

5 Evaluation

We evaluate FAST to answer four key questions:

- How does FAST compare to state-of-the-art schedulers on workloads, transfer sizes, and degrees of skew (§5.1)?
- What end-to-end throughput gains does FAST deliver to MoE training (§5.2)?
- How does FAST's scheduling runtime compare with solver-based solutions (§5.3)?
- How does FAST scale to larger clusters and varying network bandwidths (§5.4)?

Testbed. (i) *NVIDIA cluster:* 4 servers with NVIDIA H200 GPUs [44], each with 8 GPUs, interconnected by 400 Gbps InfiniBand with credit-based flow control [40] and 4 KB MTU. Intra-server scale-up uses NVLink, with a 9:1 scale-up to scale-out bandwidth ratio (450 GBps vs. 50 GBps). The scheduler runs on Intel Xeon Platinum 8468 CPUs. (ii) *AMD cluster:* 4 servers with AMD MI300X GPUs [2], each with 8 GPUs, connected via 100 Gbps RoCEv2 Ethernet with out-of-the-box DCQCN [19] as congestion control and 1 KB MTU. Intra-server scale-up is a fully connected Infinity Fabric mesh, with a 35:1 bandwidth ratio (448 GBps vs. 12.5 GBps). The scheduler runs on AMD EPYC 9534 CPUs. Each GPU has its dedicated NIC with GPU Direct RDMA [5] in both clusters.

Libraries & dependencies. We provide a Python API, `all_to_all_FAST`, mirroring PyTorch's `all_to_all_single`

for integration into existing models. Implementations of data transfers differ by hardware: (i) On H200, scale-up/scale-out uses CUDA IPC [39]/NVSHMEM [45] respectively. (ii) On MI300X, both use RCCL [6]. The transfer pipeline is implemented using multiple CUDA streams with synchronizations.

Integration into MoE systems. FAST operates in a *distributed* fashion: given the same traffic matrix, each GPU independently computes the identical global schedule, eliminating the need for a central coordinator. Only the traffic matrix—a compact integer array—must be synchronized; the schedule itself does not need to be exchanged.

This integration is natural in MoE frameworks such as Megatron-LM [54], which already materialize the per-`alltoallv` traffic matrix before each dispatch. Specifically, Megatron-LM performs an All-Gather of per-expert token counts (e.g., `num_global_tokens_per_expert` [3]), from which the full traffic matrix can be constructed. Importantly, this All-Gather is *not* introduced by FAST. It is already required by the baseline NCCL `alltoallv` implementation to compute receive counts and buffer offsets, since each GPU knows how many tokens it sends but not how many it will receive from others. FAST simply consumes this existing traffic matrix, synthesizes a schedule, and executes the transfers.

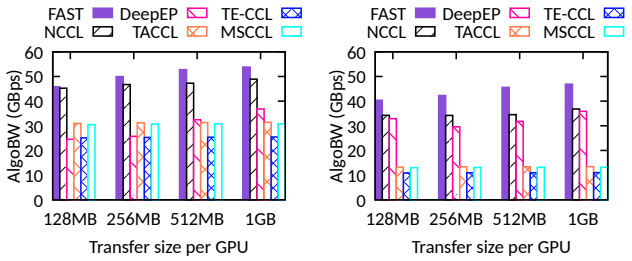
Workloads. We evaluate FAST on both synthetic and real MoE workloads. For synthetic workloads, we model skewness by varying GPU-pair transfer sizes using two distributions: (i) *random* `alltoallv` with uniformly-distributed sizes, and (ii) *skewed* `alltoallv` with Zipfian-distributed sizes. For real workloads, we focus on MoE training, where prior work identifies `alltoallv` as the primary bottleneck [22, 27, 29], and report the end-to-end throughput improvements.

We also evaluate *repetitive, balanced* All-to-All workloads, where existing schedulers can amortize their cost, enabling fair comparison in settings favoring prior approaches.

Metrics. Our primary metric is *algorithmic bandwidth*, widely used in prior work [4, 53]. It captures how fast a transfer completes, defined as $\frac{\text{Total transfer size}}{\text{\# of GPUs} \times \text{Completion Time}}$. Because skewed `alltoallv` may have variable per-GPU volumes, this average normalizes across all GPUs. Algorithmic bandwidth can exceed the raw scale-out link bandwidth, since part of the transfer completes locally over the faster scale-up fabric. For example, in a 4-node cluster with 50 GBps scale-out links, if 25% of the traffic is intra-server, the optimal algorithmic bandwidth is $50/0.75 = 66.6$ GBps. Higher is better.

Our second metric is *scheduling runtime*, which captures the time to synthesize a schedule. Lower is better.

Baselines. We compare FAST against two classes of baselines: (i) *Solver-based schedulers*: TACCL [53] and TE-CCL [31], which used constraint-based solvers for scheduling. We evaluate them on both NVIDIA and AMD clusters. (ii) *Industry libraries*: On NVIDIA: NCCL [4] (version 2.27.3), DeepEP [55] (from DeepSeek [15]), and MSCCL [14]. On AMD: RCCL [6], SpreadOut [49] (abbreviated as 'SPO'), and MSCCL. Since DeepEP is NVIDIA-only, we do not evaluate it on AMD. NCCL outperforms SpreadOut on NVIDIA, so we omit SpreadOut there; on AMD, RCCL's `alltoallv` lacks effective scheduling, making SpreadOut a stronger baseline that we include.



(a) Random (b) Skewed w/ factor 0.8

Figure 12: `alltoallv` performance on NVIDIA testbed (with 450 GBps scale-up and 50 GBps scale-out).

5.1 `alltoallv` Performance

5.1.1 Performance Under Different Transfer Sizes

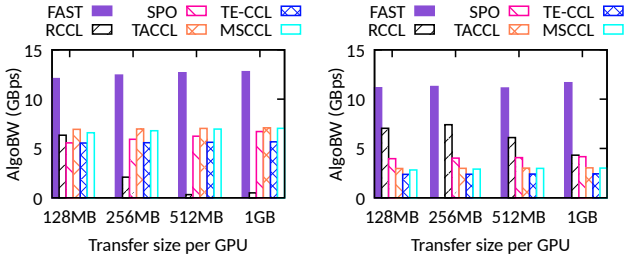
FAST consistently achieves the best `alltoallv` performance on both NVIDIA and AMD testbeds. For baselines, NCCL, DeepEP, SpreadOut, and RCCL support `alltoallv` natively, while TACCL, TE-CCL, SyCCL, and MSCCL only support balanced All-to-All. Adapting the solvers to skewed `alltoallv` can be done in two ways: (i) explicitly encoding variable flow sizes into the problem formulation, or (ii) padding all flows to a uniform size so the solver sees a balanced workload (padding data is used only for scheduling, not for actual transfers). We attempted the first approach, but it made the solvers—already slow on balanced workloads (e.g., TACCL needs over 30 minutes for 32 GPUs)—incapable of finishing within a reasonable time. Thus, we adopt padding to simplify workloads for solver-based schedulers.

We evaluate with per-GPU message sizes from 100 MB to 1 GB, representative of typical workloads reported by prior work [29, 55]. The improvement factors vary across testbeds, reflecting differences in both network hardware and software implementation, as discussed below.

Result on NVIDIA testbed. As shown in Figure 12a, FAST achieves the best algorithmic bandwidth under random workloads. It slightly outperforms NCCL by 1.01–1.1×, and exceeds DeepEP (1.5–1.9×) and TACCL (1.5–1.7×). Performance improves with larger transfers, as scale-out links saturate more easily and staging overheads in FAST are amortized.

NCCL with PXN [32] employs *sender-side aggregation*, consolidating outgoing flows at proxy GPUs before traversing scale-out links. By aggregating traffic across flows, PXN reduces per-GPU variance and mitigates mild skew. As a result, under mildly skewed workloads, NCCL can approach FAST's performance even without explicit traffic rebalancing.

DeepEP [55] places aggregation and fan-out *on the receiver side*. Data are first delivered to ingress GPUs on the destination server and then forwarded via NVLink to their target



(a) Random (b) Skewed w/ factor 0.8

Figure 13: `alltoallv` performance on AMD testbed (with 448 GBps scale-up and 12.5 GBps scale-out).

GPUs. Under skew, multiple ingress GPUs may concurrently forward large volumes to the same targets, causing NVLink receive contention and local hotspots that limit throughput—as observed by DeepEP's own NVLink runtime profiler.

Solver-based schedulers (e.g., TACCL, TE-CCL) convert skewed `alltoallv` into a fictitiously balanced All-to-All via padding, which significantly reduces synthesis time and enables practical schedule generation. However, the padded transfers do not correspond to real data movement and still occupy communication slots, delaying actual transfers. So their performances are much lower than FAST in practice.

Straggler effects intensify under skew. Under Zipfian workloads (Figure 12b), FAST outperforms NCCL by 1.2–1.3×, DeepEP by 1.2–1.5×, and TACCL by over 3×. The performance gap with NCCL widens as the workload shifts from uniform to Zipfian: even with PXN aggregation, residual imbalance introduces stragglers that limit NCCL's efficiency.

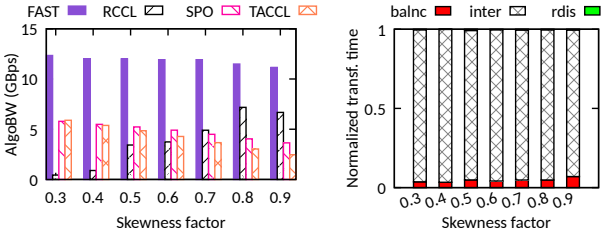
Results on AMD testbed. FAST again achieves the best performance under random workloads (Figure 13a), surpassing TACCL by 1.3–1.8×, TE-CCL by 1.6–2.3×, SpreadOut by 1.9–2.1×, and RCCL by 1.1–10×. As with NVIDIA, most algorithms benefit from larger transfers.

RCCL, however, shows the opposite trend: throughput decreases with transfer size. This is mainly due to its `alltoallv` implementation—launching all flows concurrently with no scheduling—causing severe incast and reduced goodput.

Under skewed workloads (Figure 13b), FAST extends its lead, outperforming TACCL by 2.9–3.8×, TE-CCL by 3.6–4.7×, SpreadOut by 2.5–2.8×, and RCCL by 1.3–2.6×. Interestingly, RCCL performs relatively better here than under random workloads: skew concentrates traffic into a few elephant flows while leaving most as short mice transfers, reducing widespread collisions and easing incast pressure.

5.1.2 Performance under Balanced All-to-All

On the simple, repetitive balanced workload, DeepEP (60 GBps), TACCL (59 GBps), and NCCL (58 GBps) all achieve good performance. In this setting, FAST achieves 58 GBps—slightly below the best—since its balancing and redistribution add minor overhead unnecessary when the workload is already balanced. While prior work efficiently handles balanced All-to-All, they lack mechanisms to address skew-induced stragglers, where FAST provides clear advantages.



(a) Performance (b) Breakdown

Figure 14: `alltoallv` performance and transfer time breakdown under different skewness on AMD testbed.

5.1.3 Performance under Different Skewness

We generate skewed workloads using a Zipfian distribution with varying skewness factors. A larger factor produces more mice flows and amplifies elephant flows, creating a stronger imbalance. The `alltoallv` traces we profile during MoE pretraining show skewness factors between 0.4 and 0.8.

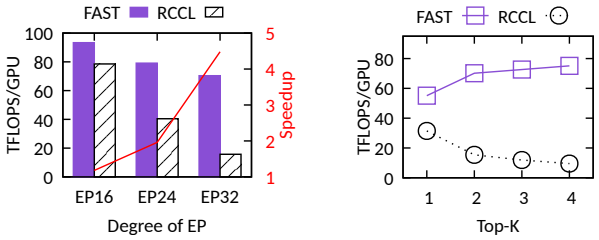
On the AMD testbed, we compare FAST with TACCL, SpreadOut, and RCCL across different skewness levels (Figure 14a). FAST consistently delivers the best performance, outperforming RCCL by 1.6–10×, SpreadOut by 2.1–3.1×, and TACCL by 2.1–4.5× (TE-CCL omitted here as it performs slightly worse than TACCL).

The performance gap reflects how each system handles stragglers. (i) For FAST, increased skew lengthens the balancing phase, but the overhead remains modest as shown in Figure 14b: even at skew factor 0.9, balancing and redistribution account for under 8% of scale-out time (under 5% in most cases). Since scale-out dominates and runs at full efficiency, FAST remains within 1.08× of optimal. (ii) TACCL degrades under skew because heavier skew requires more padding, reducing effective efficiency. (iii) SpreadOut suffers as skew amplifies per-stage imbalance, causing overall transfer time to be dominated by stragglers. (iv) RCCL shows the opposite trend: higher skew produces more mice flows, whose contention is absorbed by switch buffers, reducing incast severity and improving performance on the AMD testbed.

5.2 End-to-End Performance

To evaluate FAST in an end-to-end setting, we integrate it into Megatron-LM [54] on the AMD testbed to perform on-the-fly scheduling for every `alltoallv` communication during MoE training. We compare against PyTorch's [47] default `all_to_all_single` operator, which uses RCCL as the backend. Solver-based approaches cannot be integrated due to their prohibitive scheduling overhead.

We vary two key MoE configurations to study how FAST behaves under different training scenarios: (i) *Expert parallelism (EP)*: We sweep EP from 16 to 24 to 32, which directly determines the scale of `alltoallv`. Under the configuration where each GPU hosts one expert (like DeepSeek [15]), this corresponds to scaling the transfer from 16 GPUs (2 servers) to 32 GPUs (4 servers). (ii) *Top-K routing*: In MoE, each input token is routed to the Top-K most relevant experts;



(a) Vary EP (b) Vary Top-K routing

Figure 15: Megatron-LM MoE training performance improvement on AMD testbed.

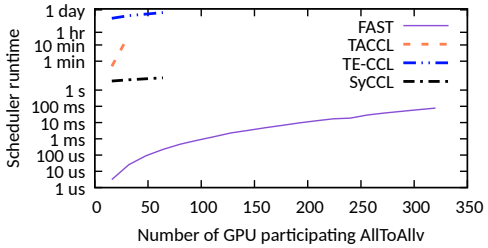


Figure 16: Comparison of FAST 's scheduling runtime against state-of-the-art solver-based schedulers (log-scale).

larger K increases token replication and thus flow size in the `alltoallv` workload. For this experiment, we fix EP at 32 and vary K. Other workload-related parameters, such as batch size and sequence length, similarly affect communication size and training performance.

As shown in Figure 15a, FAST delivers a 1.18–4.48× speedup in end-to-end training throughput (under Top-2 routing) across different EP levels. Two main trends emerge: (i) Training throughput (left y-axis) decreases as EP increases. This is expected since higher EP involves more GPUs, more servers, and thus more scale-out traffic, which lowers communication efficiency and increases GPU idle time. (ii) The baseline degrades sharply as EP grows due to escalating incast. RCCL performs no scheduling across transfers, leaving congestion entirely to the transport layer. For example, with EP16, a receiver GPU/NIC handles up to 8 concurrent flows, while with EP32 this rises to 24. With out-of-the-box DCQCN as congestion control, this causes severe throughput collapse.

As shown in Figure 15b, FAST outperforms the baseline by 1.75–7.88×. Notably, FAST and RCCL exhibit opposite scaling with K: increasing K improves FAST by enlarging flows and amortizing staging overhead, but degrades RCCL due to increased flow collisions and congestion.

5.3 Scheduling Overhead

FAST introduces two types of overhead relative to non-scheduling algorithms such as NCCL: (i) additional scheduling runtime, and (ii) extra memory for intermediate buffers.

Scheduling runtime. As shown in Figure 16, FAST scales to 320 GPUs with only 77 ms of overhead—faster than SyCCL, the fastest solver-based scheduler, which already takes 3.6s at just 16 GPUs. Earlier solver-based methods generally fail to scale beyond 64 GPUs, rendering them unusable for moderate expert-parallelism levels such as EP96 and EP128.

Even at smaller scales, their scheduling time remains prohibitively long—ranging from seconds to hours—far exceeding the transfer time itself and much longer than the interval before the workload itself changes. In comparison, FAST 's lightweight scheduling enables *on-the-fly* planning.

Memory overhead. FAST also requires additional memory for temporary buffers that hold rebalanced or redistributed data. Under random workloads, this overhead is about ≈30% of the original `alltoallv` buffer size. In practice, the impact is minimal: typical `alltoallv` buffers are under 1 GB (e.g., in DeepEP [55]), so the extra cost is less than 300 MB. On modern GPUs such as the NVIDIA H200 [44] with 141 GB of memory, this represents under 0.22% of total capacity—an acceptable tradeoff for the performance gains.

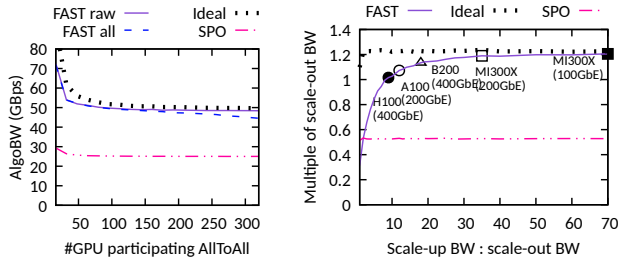
5.4 Scaling and Bandwidth Sensitivity

We use simulation to evaluate FAST beyond the limits of our physical testbeds, exploring both larger scales and different scale-up/scale-out bandwidth configurations. The simulator follows the analytical framework widely used in prior work such as TE-CCL and TACCL [31,53]: given a schedule with a sequence of transfer steps (each with a defined size), the completion time is computed by summing per-step costs. Each cost consists of a fixed link wake-up delay plus the transmission time ($\frac{\text{data size}}{\text{link bandwidth}}$).

We focus on scenarios that solver-based schedulers cannot scale to and therefore exclude them from comparison. Accordingly, we compare FAST against SpreadOut and an optimal bandwidth bound, which assumes infinitely fast scale-up links so that intra-server transfers are instantaneous. Under this bound, scale-out is the only bottleneck, and the optimal time is defined by the maximum balanced sender or receiver load divided by the scale-out bandwidth.

Performance at larger scale. We first scale the number of GPUs in `alltoallv`, with each GPU pair transmitting 50 MB on average in a random workload, simulated in a 400 Gbps scale-out network and a 450 GBps scale-up network (H200). As shown in Figure 17a, FAST stays within 5% of optimal when scheduling time is excluded ("FAST raw"). Including scheduling time, the gap widens to 10% at larger scales, since scheduling cost grows faster than workload completion time (which increases only linearly with GPU count). We leave on-the-fly scheduling at extreme scale as future work. By contrast, SpreadOut achieves only about half of FAST 's throughput.

Performance under varying scale-up/scale-out ratios. We also evaluate FAST under different scale-up/scale-out bandwidth ratios on a 32-GPU setup. As shown in Figure 17b, normalized bandwidth is reported relative to scale-out capacity, which can exceed 1 since about 25% of traffic is intra-server, giving an upper bound near 1.25. Performance improves as the scale-up/scale-out ratio increases, as faster scale-up links further reduce balancing and redistribution overhead. These



(a) Perf. at scale (b) Varying bandwidth ratios

Figure 17: Simulation of FAST at larger scales and varying scale-up/scale-out bandwidth ratios under random workloads.

results indicate that FAST sustains high efficiency on modern GPUs with advanced scale-up interconnects.

6 Discussion

Other collectives. FAST is intentionally specialized for `alltoallv`, where skewness and dynamism fundamentally limit existing approaches, and is not designed as a general-purpose collective scheduler. For balanced collectives such as All-Reduce and All-Gather, communication patterns are static with uniform flow sizes, and near-optimal schedules can typically be achieved using existing library or solver-based implementations; a dynamic, traffic-aware scheduler provides little additional benefit. In practice, modern libraries (e.g., NCCL) already select among multiple collective algorithms at runtime, and FAST naturally fits this model as a specialized implementation for skewed `alltoallv` workloads.

Other topologies. FAST targets two-tier GPU cluster topologies that combine scale-up and scale-out fabrics, which are increasingly common in modern deployments [18, 38] with HGX-style platforms [46]. The core insight of FAST is to reshape traffic to reduce skew before it reaches slower network tiers, thereby simplifying the workload imposed on bottleneck fabric. While our current implementation focuses on two tiers, this principle naturally extends to multi-tier networks.

Hybrid parallelism. Our evaluation focuses on pure expert parallelism (EP). In hybrid deployments combining EP with tensor or pipeline parallelism, FAST can be applied directly when collectives execute in disjoint time windows. Coordinating FAST with concurrent collectives that share networks introduces additional complexity and is left for future work.

Persistence of scale-up vs. scale-out bandwidth gaps. We expect the bandwidth gap between scale-up and scale-out fabrics to persist, as scale-up interconnects benefit from short distances and tight hardware–software co-design, while scale-out networks must span longer distances and support backward compatibility and multi-tenancy. Exploiting scale-up bandwidth will therefore remain an important opportunity.

Interaction with congestion control. FAST operates at the collective layer and is orthogonal to transport-layer congestion control. Even under ideal congestion control, FAST can provide additional benefits by reducing traffic skew and improving NIC utilization, preventing communication from being bottlenecked by a small number of heavily loaded NICs.

7 Related Work

Collective communication schedulers. Classic All-to-All algorithms such as SpreadOut [37] assume balanced workloads and single-tier networks. Modern GPU libraries (e.g., NCCL [4], RCCL [6]) exploit two-tier fabrics but do not handle skew, stragglers, or incast. Solver-based schedulers such as SCCL, TACCL, TE-CCL, and SyCCL [10, 11, 31, 53] can produce near-optimal schedules but incur prohibitive overheads, making them unsuitable for dynamic workloads. In contrast, FAST provides a scalable, on-the-fly scheduler for skewed and dynamic `alltoallv` in modern GPU clusters.

MoE optimizations. Prior work on mixture-of-experts training [15, 17, 22, 23, 27, 28, 50] improves efficiency through model design or communication–computation overlap, but typically treats `alltoallv` as a black box. FAST complements these efforts by directly optimizing `alltoallv`.

Leveraging NIC idleness. Recent work, FuseLink [51], exploits under-utilized NIC bandwidth but operates below the collective layer, preserving the logical communication structure. Although FuseLink shifts traffic onto idle NICs, it does not reassign flow pairings and thus cannot address bottlenecks from skew or incast. FAST instead reshapes communication using overlay paths that stage transfers through intermediate GPUs, explicitly mitigating hotspots and contention.

Switch designs. Prior work applies Birkhoff's decomposition to switch scheduling [12, 30] and circuit-switched All-to-All communication [52]. FAST instead applies it at the GPU collective layer, running over commodity packet-switched networks without any switch changes.

8 Conclusion

All-to-All communication is critical to modern distributed systems. We present FAST, the first polynomial-time, on-the-fly scheduler for skewed and dynamic `alltoallv`. FAST absorbs skew using fast scale-up links and enforces balanced one-to-one transfers over scale-out, enabling efficient `alltoallv` communication. Across NVIDIA and AMD testbeds, FAST outperforms state-of-the-art systems while reducing synthesis time by orders of magnitude.

9 Acknowledgments

We thank our shepherd, Kai Chen, and the anonymous reviewers for their constructive feedback. We also thank Yonghao Zhuang, Zhihao Jia, Isabel Suizo, and Hugo Sadok for feedback on the drafts of this work. This work was supported by ONR Award N000142412059 and a Sloan Research Fellowship. Justine Sherry holds concurrent appointments as an Associate Professor at CMU and as an Amazon Scholar. This work was conducted at CMU and is not affiliated with Amazon. This work 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.

References

[1] Amd cdna2 architecture. https://www.amd.com/content/dam/amd/en/documents/instinct-business-docs/white-papers/amd-cdna2-white-paper.pdf.

[2] Amd cdna3 architecture. https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/white-papers/amd-cdna-3-white-paper.pdf.

[3] Megatron-lm's allgather operation before alltoall. https://github.com/NVIDIA/Megatron-LM/blob/main/megatron/core/transformer/moe/token_dispatcher.py.

[4] Nvidia collective communications library (nccl). https://developer.nvidia.com/nccl.

[5] Nvidia gpudirect. https://developer.nvidia.com/gpudirect.

[6] Rocm communication collectives library (rccl). https://github.com/ROCm/rccl.

[7] AI, P. Efficient and portable mixture-of-experts communication. https://www.perplexity.ai/hub/blog/efficient-and-portable-mixture-of-experts-communication, 2025.

[8] Ayala, A., Tomov, S., Luo, X., Shaeik, H., Haidar, A., Bosilca, G., and Dongarra, J. Impacts of multi-gpu mpi collective communications on large fft computation. In *2019 IEEE/ACM Workshop on Exascale MPI (ExaMPI)* (2019), pp. 12–18.

[9] Birkhoff, G. Three observations on linear algebra. *Univ. Nac. Tucumán. Revista A. 5* (1946), 147–151.

[10] Cai, Z., Liu, Z., Maleki, S., Musuvathi, M., Mytkowicz, T., Nelson, J., and Saarikivi, O. Synthesizing optimal collective algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Feb. 2021), ACM.

[11] Cao, J., Shi, S., Gao, J., Liu, W., Yang, Y., Xu, Y., Zheng, Z., Guan, Y., Qian, K., Liu, Y., Xu, M., Wang, T., Wang, N., Dong, J., Fu, B., Cai, D., and Zhai, E. Syccl: Exploiting symmetry for efficient collective communication scheduling. In *Proceedings of the ACM SIGCOMM 2025 Conference* (New York, NY, USA, 2025), SIGCOMM '25, Association for Computing Machinery, p. 645–662.

[12] Chang, C.-S., Chen, W.-J., and Huang, H.-Y. Birkhoff-von neumann input buffered crossbar switches. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No.00CH37064)* (2000), vol. 3, pp. 1614–1623 vol.3.

[13] Consortium, U. E. Overview of and Motivation for the Forthcoming Ultra Ethernet Consortium Specification. Tech. rep., 2023.

[14] Cowan, M., Maleki, S., Musuvathi, M., Saarikivi, O., and Xiong, Y. Mscclang: Microsoft collective communication language. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (New York, NY, USA, 2023), ASPLOS 2023, Association for Computing Machinery, p. 502–514.

[15] DeepSeek-AI, Liu, A., Feng, B., Xue, B., Wang, B., Wu, B., Lu, C., Zhao, C., Deng, C., Zhang, C., Ruan, C., Dai, D., Guo, D., Yang, D., Chen, D., Ji, D., Li, E., Lin, F., Dai, F., Luo, F., Hao, G., Chen, G., Li, G., Zhang, H., Bao, H., Xu, H., Wang, H., Zhang, H., Ding, H., Xin, H., Gao, H., Li, H., Qu, H., Cai, J. L., Liang, J., Guo, J., Ni, J., Li, J., Wang, J., Chen, J., Chen, J., Yuan, J., Qiu, J., Li, J., Song, J., Dong, K., Hu, K., Gao, K., Guan, K., Huang, K., Yu, K., Wang, L., Zhang, L., Xu, L., Xia, L., Zhao, L., Wang, L., Zhang, L., Li, M., Wang, M., Zhang, M., Zhang, M., Tang, M., Li, M., Tian, N., Huang, P., Wang, P., Zhang, P., Wang, Q., Zhu, Q., Chen, Q., Du, Q., Chen, R. J., Jin, R. L., Ge, R., Zhang, R., Pan, R., Wang, R., Xu, R., Zhang, R., Chen, R., Li, S. S., Lu, S., Zhou, S., Chen, S., Wu, S., Ye, S., Ye, S., Ma, S., Wang, S., Zhou, S., Yu, S., Zhou, S., Pan, S., Wang, T., Yun, T., Pei, T., Sun, T., Xiao, W. L., Zeng, W., Zhao, W., An, W., Liu, W., Liang, W., Gao, W., Yu, W., Zhang, W., Li, X. Q., Jin, X., Wang, X., Bi, X., Liu, X., Wang, X., Shen, X., Chen, X., Zhang, X., Chen, X., Nie, X., Sun, X., Wang, X., Cheng, X., Liu, X., Xie, X., Liu, X., Yu, X., Song, X., Shan, X., Zhou, X., Yang, X., Li, X., Su, X., Lin, X., Li, Y. K., Wang, Y. Q., Wei, Y. X., Zhu, Y. X., Zhang, Y., Xu, Y., Xu, Y., Huang, Y., Li, Y., Zhao, Y., Sun, Y., Li, Y., Wang, Y., Yu, Y., Zheng, Y., Zhang, Y., Shi, Y., Xiong, Y., He, Y., Tang, Y., Piao, Y., Wang, Y., Tan, Y., Ma, Y., Liu, Y., Guo, Y., Wu, Y., Ou, Y., Zhu, Y., Wang, Y., Gong, Y., Zou, Y., He, Y., Zha, Y., Xiong, Y., Ma, Y., Yan, Y., Luo, Y., You, Y., Liu, Y., Zhou, Y., Wu, Z. F., Ren, Z. Z., Ren, Z., Sha, Z., Fu, Z., Xu, Z., Huang, Z., Zhang, Z., Xie, Z., Zhang, Z., Hao, Z., Gou, Z., Ma, Z., Yan, Z., Shao, Z., Xu, Z., Wu, Z., Zhang, Z., Li, Z., Gu, Z., Zhu, Z., Liu,

Z., LI, Z., XIE, Z., SONG, Z., GAO, Z., AND PAN, Z. Deepseek-v3 technical report, 2024.

[16] DUFOSSÉ, F., AND UÇAR, B. Notes on birkhoff–von neumann decomposition of doubly stochastic matrices. *Linear Algebra and its Applications 497* (2016), 108–115.

[17] FEDUS, W., ZOPH, B., AND SHAZEER, N. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity, 2022.

[18] GANGIDI, A., MIAO, R., ZHENG, S., BONDU, S. J., GOES, G., MORSY, H., PURI, R., RIFTADI, M., SHETTY, A. J., YANG, J., ZHANG, S., FERNANDEZ, M. J., GANDHAM, S., AND ZENG, H. Rdma over ethernet for distributed training at meta scale. In *Proceedings of the ACM SIGCOMM 2024 Conference* (New York, NY, USA, 2024), ACM SIGCOMM '24, Association for Computing Machinery, p. 57–70.

[19] GAO, Y., YANG, Y., CHEN, T., ZHENG, J., MAO, B., AND CHEN, G. Dcqcn+: Taming large-scale incast congestion in rdma over ethernet networks. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)* (2018), pp. 110–120.

[20] GUROBI OPTIMIZATION, LLC. Gurobi Optimizer Reference Manual, 2024.

[21] HOEFLER, T., SCHRAMM, K., SPADA, E., UNDERWOOD, K., ALEXANDER, C., ALVERSON, B., BOTTORFF, P., CAULFIELD, A., HANDLEY, M., HUANG, C., RAICIU, C., KABBANI, A., OPSASNICK, E., PAN, R., RAN, A., AND SOHAN, R. Ultra ethernet's design principles and architectural innovations, 2025.

[22] HWANG, C., CUI, W., XIONG, Y., YANG, Z., LIU, Z., HU, H., WANG, Z., SALAS, R., JOSE, J., RAM, P., CHAU, J., CHENG, P., YANG, F., YANG, M., AND XIONG, Y. Tutel: Adaptive mixture-of-experts at scale, 2023.

[23] JIANG, A. Q., SABLAYROLLES, A., ROUX, A., MENSCH, A., SAVARY, B., BAMFORD, C., CHAPLOT, D. S., DE LAS CASAS, D., HANNA, E. B., BRESSAND, F., LENGYEL, G., BOUR, G., LAMPLE, G., LAVAUD, L. R., SAULNIER, L., LACHAUX, M.-A., STOCK, P., SUBRAMANIAN, S., YANG, S., ANTONIAK, S., SCAO, T. L., GERVET, T., LAVRIL, T., WANG, T., LACROIX, T., AND SAYED, W. E. Mixtral of experts, 2024.

[24] JOHNSON, D. M., DULMAGE, A. L., AND MENDELSOHN, N. S. On an algorithm of g. birkhoff concerning doubly stochastic matrices. *Canadian Mathematical Bulletin 3*, 3 (1960), 237–242.

[25] KUHN, H. W. The Hungarian Method for the Assignment Problem. *Naval Research Logistics Quarterly 2*, 1–2 (March 1955), 83–97.

[26] LEI, Y., ET AL. Github repo for "FAST: An efficient scheduler for all-to-all gpu communication". https://github.com/MangoBoost/FAST, 2026.

[27] LEPIKHIN, D., LEE, H., XU, Y., CHEN, D., FIRAT, O., HUANG, Y., KRIKUN, M., SHAZEER, N., AND CHEN, Z. Gshard: Scaling giant models with conditional computation and automatic sharding, 2020.

[28] LI, J., JIANG, Y., ZHU, Y., WANG, C., AND XU, H. Accelerating distributed MoE training and inference with lina. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)* (Boston, MA, July 2023), USENIX Association, pp. 945–959.

[29] LIAO, X., SUN, Y., TIAN, H., WAN, X., JIN, Y., WANG, Z., REN, Z., HUANG, X., LI, W., TSE, K. F., ZHONG, Z., LIU, G., ZHANG, Y., YE, X., ZHANG, Y., AND CHEN, K. Mixnet: A runtime reconfigurable optical-electrical fabric for distributed mixture-of-experts training, 2025.

[30] LIU, H., MUKERJEE, M. K., LI, C., FELTMAN, N., PAPEN, G., SAVAGE, S., SESHAN, S., VOELKER, G. M., ANDERSEN, D. G., KAMINSKY, M., PORTER, G., AND SNOEREN, A. C. Scheduling techniques for hybrid circuit/packet networks. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies* (New York, NY, USA, 2015), CoNEXT '15, Association for Computing Machinery.

[31] LIU, X., ARZANI, B., KAKARLA, S. K. R., ZHAO, L., LIU, V., CASTRO, M., KANDULA, S., AND MARSHALL, L. Rethinking machine learning collective communication as a multi-commodity flow problem. In *Proceedings of the ACM SIGCOMM 2024 Conference* (New York, NY, USA, 2024), ACM SIGCOMM '24, Association for Computing Machinery, p. 16–37.

[32] MANDAKOLATHUR, K., AND JEAUGEY, S. 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.

[33] MITTAL, R., LAM, V. T., DUKKIPATI, N., BLEM, E., WASSEL, H., GHOBADI, M., VAHDAT, A., WANG, Y., WETHERALL, D., AND ZATS, D. Timely: Rtt-based congestion control for the datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, Association for Computing Machinery, p. 537–550.

[34] MPI. MPI_Alltoallv, 2024.

[35] MUDIGERE, D., HAO, Y., HUANG, J., JIA, Z., TULLOCH, A., SRIDHARAN, S., LIU, X., OZDAL, M., NIE, J., PARK, J., LUO, L., YANG, J. A., GAO, L., IVCHENKO, D., BASANT, A., HU, Y., YANG, J., ARDESTANI, E. K., WANG, X., KOMURAVELLI, R., CHU, C.-H., YILMAZ, S., LI, H., QIAN, J., FENG, Z., MA, Y., YANG, J., WEN, E., LI, H., YANG, L., SUN, C., ZHAO, W., MELTS, D., DHULIPALA, K., KISHORE, K., GRAF, T., EISENMAN, A., MATAM, K. K., GANGIDI, A., CHEN, G. J., KRISHNAN, M., NAYAK, A., NAIR, K., MUTHIAH, B., KHORASHADI, M., BHATTACHARYA, P., LAPUKHOV, P., NAUMOV, M., MATHEWS, A., QIAO, L., SMELYANSKIY, M., JIA, B., AND RAO, V. Software-hardware co-design for fast and scalable training of deep learning recommendation models, 2023.

[36] NAUMOV, M., KIM, J., MUDIGERE, D., SRIDHARAN, S., WANG, X., ZHAO, W., YILMAZ, S., KIM, C., YUEN, H., OZDAL, M., NAIR, K., GAO, I., SU, B.-Y., YANG, J., AND SMELYANSKIY, M. Deep learning training in facebook data centers: Design of scale-up and scale-out systems, 2020.

[37] NETTERVILLE, N., FAN, K., KUMAR, S., AND GILRAY, T. A visual guide to mpi all-to-all. In *2022 IEEE 29th International Conference on High Performance Computing, Data and Analytics Workshop (HiPCW)* (2022), pp. 20–27.

[38] NETWORKS, J. Networking the ai data center, 2024. https://www.juniper.net/content/dam/www/assets/white-papers/us/en/networking-the-ai-data-center.pdf.

[39] NICKOLLS, J., BUCK, I., GARLAND, M., AND SKADRON, K. Scalable parallel programming with cuda. In *ACM SIGGRAPH 2008 Classes* (New York, NY, USA, 2008), SIGGRAPH '08, Association for Computing Machinery.

[40] NVIDIA. Infiniband credit-based flow control, 2005. https://network.nvidia.com/pdf/whitepapers/deploying_qos_wp_10_19_2005.pdf.

[41] NVIDIA. Nvidia v100 gpu architecture, 2017. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[42] NVIDIA. Connectx-8 supernic. https://resources.nvidia.com/en-us-accelerated-networking-resource-library/connectx-datasheet-c, 2025.

[43] NVIDIA. Nvidia h100 gpu architecture, 2025. https://resources.nvidia.com/en-us-tensor-core.

[44] NVIDIA. Nvidia h200 gpu, 2025. https://www.nvidia.com/en-us/data-center/h200/.

[45] NVIDIA. Nvidia openshmem library, 2025. https://docs.nvidia.com/nvshmem/api/index.html.

[46] NVIDIA. Nvidia hgx platform, 2026. https://www.nvidia.com/en-us/data-center/hgx/.

[47] PASZKE, A., GROSS, S., MASSA, F., LERER, A., BRADBURY, J., CHANAN, G., KILLEEN, T., LIN, Z., GIMELSHEIN, N., ANTIGA, L., DESMAISON, A., KÖPF, A., YANG, E., DEVITO, Z., RAISON, M., TEJANI, A., CHILAMKURTHY, S., STEINER, B., FANG, L., BAI, J., AND CHINTALA, S. Pytorch: An imperative style, high-performance deep learning library, 2019.

[48] PEKUROVSKY, D. P3dfft: A framework for parallel computations of fourier transforms in three dimensions. *SIAM Journal on Scientific Computing 34*, 4 (Jan. 2012), C192–C209.

[49] PJESIVAC-GRBOVIC, J., ANGSKUN, T., BOSILCA, G., FAGG, G., GABRIEL, E., AND DONGARRA, J. Performance analysis of mpi collective operations. In *19th IEEE International Parallel and Distributed Processing Symposium* (2005), pp. 8 pp.–.

[50] RAJBHANDARI, S., LI, C., YAO, Z., ZHANG, M., AMINABADI, R. Y., AWAN, A. A., RASLEY, J., AND HE, Y. Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale, 2022.

[51] REN, Z., LI, Y., WANG, Z., HUANG, X., LI, W., XU, K., LIAO, X., SUN, Y., LIU, B., TIAN, H., ZHANG, J., WANG, M., ZHONG, Z., LIU, G., ZHANG, Y., AND CHEN, K. Enabling efficient gpu communication over multiple nics with fuselink. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation* (USA, 2025), OSDI '25, USENIX Association.

[52] RENGANATHAN, S., AND MCKEOWN, N. Chronos: Prescheduled circuit switching for llm training. In *Proceedings of the 2nd Workshop on Networks for AI Computing* (New York, NY, USA, 2025), NAIC '25, Association for Computing Machinery, p. 89–97.

[53] SHAH, A., CHIDAMBARAM, V., COWAN, M., MALEKI, S., MUSUVATHI, M., MYTKOWICZ, T., NELSON, J., SAARIKIVI, O., AND SINGH, R. Taccl: Guiding collective algorithm synthesis using communication sketches, 2022.

[54] SHOEYBI, M., PATWARY, M., PURI, R., LEGRESLEY, P., CASPER, J., AND CATANZARO, B. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[55] ZHAO, C., ZHOU, S., ZHANG, L., DENG, C., XU, Z., LIU, Y., YU, K., LI, J., AND ZHAO, L. Deepep: an efficient expert-parallel communication library. https://github.com/deepseek-ai/DeepEP, 2025.

[56] ZHAO, H., WENG, H., LU, D., LI, A., LI, J., PANDA, A., AND XIE, S. On scaling up 3d gaussian splatting training, 2024.

A Appendix

A.1 Performance Bound under Adversarial Workload

We establish a theoretical bound on FAST's performance by analyzing its behavior under adversarial workloads that trigger its worst-case execution.

We first introduce the symbols and assumptions used for proofs. There are n servers and m GPUs and m NICs within each server, making a total of $m \times n$ GPUs participate in All-to-All. We denote the per-GPU bandwidth of scale-up and scale-out network as B_1 and B_2. The scale-up network topology is a switch, while the other topology's performance bound can be derived in a similar way. The total transfer size between two different servers i and j is denoted as T_{ij} while intra-server portion of All-to-All in server i is denoted as S_i. T_{ij} $(i \neq j)$ and S_i constitutes the complete All-to-All transfer workload. Note that T_{ii} does not represent anything and is thus set to be zero for the ease of writing proofs. We don't prove the situation where the intra-node transfer is the majority of the All-to-All workload because there are more scale-out pairs than scale-up pairs in multi-node All-to-All workload, making this scenario rare. So, we assume each server's intra-node transfer size is no larger than the average of all the inter-node transfers, i.e., $S_i \leq \frac{1}{n}\sum_{j=0}^{n-1}(T_{ij})$.

Theorem 1. *The optimal transfer completion time $t_{optimal}$ is:*

$$\frac{1}{mB_2}\max(\max_{i=0}^{n-1}(\sum_{j=0}^{n-1}T_{ij}),\max_{j=0}^{n-1}(\sum_{i=0}^{n-1}T_{ij}))$$

.

PROOF. *Let us compute the transfer completion time in an ideal world where the bandwidth of an intra-node network is infinite. The intra-node transfer S_i, load balancing, and data redistribution can therefore be instantly completed. After load balancing, each GPU has T_{ij}/m amount of data waiting to be transferred via inter-server links. The inter-server transfers are bound by the largest senders or receivers among all the servers, which is* $\max(\max_{i=0}^{n-1}(\sum_{j=0}^{n-1}\frac{T_{ij}}{m}),\ \max_{j=0}^{n-1}(\sum_{i=0}^{n-1}\frac{T_{ij}}{m}))$. *So, the shortest transfer completion time is the value shown in the theorem. Any real-world transfers must be slower or equal to this ideal transfer.* □

Theorem 2. *FAST's transfer worst-case completion time t_{FAST} under the adversarial workload is:*

$$\begin{aligned} t_{FAST} = {} & \frac{1}{mB_2}\max\left(\max_{i=0}^{n-1}\sum_{j=0}^{n-1}T_{ij},\max_{j=0}^{n-1}\sum_{i=0}^{n-1}T_{ij}\right) \\ & + \frac{m-1}{mB_1}\max_{i=0}^{n-1}\sum_{j=0}^{n-1}T_{ij} + \frac{1}{nB_1}\max_{i=0}^{n-1}\sum_{j=0}^{n-1}T_{ij} \\ & + \frac{1}{mB_1}\max_{i,j=0}^{n-1}T_{ij}. \end{aligned} \tag{1}$$

PROOF. *We compute FAST's performance under adversarial workload by summing the worst-case transfer time of each transfer step:* `balance` → `intra-server portion of All-to-All` → `Birkhoff's stages` → `Final stage's redistribution`.

For load balancing, it would take the longest time to complete the job when T_{ij} is located at a single GPU in the beginning, as it causes the largest amount of data (i.e., $\frac{m-1}{m}\cdot T_{ij}$) to be balanced. For a specific server pair, it takes the source GPU $\frac{m-1}{m}\cdot T_{ij}\cdot\frac{1}{B_1}$ amount of time to balance data. Among all the server pairs, this balancing step takes $t_0 = \max_{i=0}^{n-1}(\sum_{j=0}^{n-1}T_{ij})\frac{m-1}{mB_1}$.

For the intra-server portion of All-to-All, the worst case is that all the S_i data is moved between only two GPUs, leaving the rest of the scale-up network idle. So, the worst-case time among all the servers is $t_1 = \max_{i=0}^{n-1}\frac{S_i}{B_1} \leq \frac{1}{nB_1}\max_{i=0}^{n-1}(\sum_{j=0}^{n-1}T_{ij})$ by using the assumption $S_i \leq \frac{1}{n}\sum_{j=0}^{n-1}(T_{ij})$.

For staged inter-server transfer, Birkhoff's Theorem can generate at most n^2-2n+2 transfer steps. We first sort them in ascending order based on each step's transfer size and get the sorted size $l_0 \leq l_1 \leq \cdots \leq l_{n^2-2n+1}$. We then execute the transfer steps in the sorted order, which successfully hides the current step's data redistribution from the next step's inter-server transfer, since (i) the redistribution time cost of stage i is $\frac{(m-1)l_i}{B_1}$ because each GPU at destination server receives l_i amount of data from scale-out, all of which needs to be forwarded to a single GPU (worst case scenario); (ii) the scale-out transfer cost of stage $i+1$ is $\frac{l_{i+1}}{B_2}$; and (iii) $\frac{(m-1)l_i}{B_1} < \frac{l_{i+1}}{B_2}$, because $l_i \leq l_{i+1}$ and B_1 (e.g., 450 GBps in H100) is more than $(m-1)=7$ times faster than B_2 (e.g., 50 GBps) under today's $m=8$ cluster. This means staged scale-out transfers from Birkhoff's theorem are consecutive, making the actual scale-out transfer time t_2 equals $t_{optimal}$ from the ideal setting because Birkhoff let the bottleneck servers keep transmitting.

Finally, for the last stage's redistribution, since each stage does a one-to-one server matching, the worst case is when the last stage's scale-out transfer picks the largest transfer size among all the server pairs, which is $\max_{i,j=0}^{n-1}T_{ij}/m$, making the worst-case completion time as $t_3 = \frac{1}{B_1}\max_{i,j=0}^{n-1}\frac{T_{ij}}{m}$.

Therefore, FAST worst-case transfer time under adversarial workload is $t_{FAST} = t_0+t_1+t_2+t_3$, which is the value shown

in the theorem.

□

With optimal performance and FAST's worst-case performance, we can calculate the performance bound.

Theorem 3. *The gap between FAST's worst-cast performance and optimal performance is bound by* $\frac{B_2}{B_1}(m+\frac{m}{n})$.

PROOF. *We divide FAST worst-case transfer time by ideal transfer time as follows:* $\frac{t_{FAST}}{t_{optimal}} = \frac{t_0+t_1+t_2+t_3}{t_{optimal}} \leq 1+\frac{B_2}{B_1}(m+\frac{m}{n})$ *where we shrink the denominator as follows:*

$$\max(\max_{i=0}^{n-1}(\sum_{j=0}^{n-1} T_{ij}), \max_{j=0}^{n-1}(\sum_{i=0}^{n-1} T_{ij})) \geq \max_{i=0}^{n-1}(\sum_{j=0}^{n-1} T_{ij}) \geq \max_{i,j=0}^{n-1} T_{ij}$$

to cancel out the numerator and get the final result. □

In conclusion, under adversarial workloads, the worst-case performance gap of FAST relative to optimal is bounded by the scale-up to scale-out bandwidth ratio. With today's hardware—for example, a 4-node cluster with 450 GBps scale-up on H100 [43] and 400 Gbps scale-out—this bound implies that FAST's worst-case scenario completes within 2.12× of the theoretical optimum. In practice, this worst-case adversarial workload rarely happens and the performance is much closer to optimal as we show in the evaluation.