Skew-aware Adaptive All-to-allv Algorithms for Dynamic Deep Learning Workloads

Cunyang Wei
University of Maryland, College Park
College Park, Maryland, USA
cunyang@umd.edu

Abhinav Bhatele
University of Maryland, College Park
College Park, Maryland, USA
bhatele@cs.umd.edu

Abstract

All-to-allv is a commonly used collective and can be a significant performance bottleneck in high performance computing and distributed deep learning (DL) workloads. DL workloads, in particular, have significantly skewed distributions of data sizes exchanged between processes, dynamic changes in communication, and large message sizes. These make optimizing the all-to-allv communication challenging. In this work, we present SABRE, a **S**kew-aware **A**ll-to-allv library for **B**alancing ir**RE**gular communication on GPU-based clusters. SABRE is designed to achieve good performance under both highly-skewed and lightly-skewed communication patterns. It selects different optimization strategies in these regimes to reduce all-to-allv communication time on modern GPU clusters. We implement and evaluate SABRE on the Perlmutter system at NERSC, and demonstrate performance speedups in real workloads through communication imbalance detection, adaptive algorithm selection, careful backend selection, and memory management optimizations. Our library achieves up to 2.4× speedup over Cray MPICH and NCCL in microbenchmark experiments, and improves mixture of experts model training time by up to 1.8× compared with the default PyTorch implementation.

CCS Concepts

• **Computing methodologies** → **Distributed artificial intelligence**; **Parallel algorithms**.

Keywords

collective communication, all-to-allv, distributed deep learning

ACM Reference Format:
Cunyang Wei and Abhinav Bhatele. 2026. Skew-aware Adaptive All-to-allv Algorithms for Dynamic Deep Learning Workloads. In *2026 International Conference on Supercomputing (ICS '26), July 06–09, 2026, Belfast, United Kingdom.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3797905.3800541

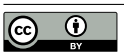
1 Introduction

All-to-all communication between processes in a parallel program is a fundamental collective operation where each process exchanges a distinct data block of the same size with every other process. The all-to-allv operation extends this pattern by allowing data blocks of different sizes between process pairs, which is essential for workloads with irregular data distributions.

In high performance computing (HPC), many workloads rely on 3D Fast Fourier Transforms (FFT) [18], where all-to-all(v) operations are the main bottleneck during transpose phases. In deep learning, mixture of experts (MoE) models such as GShard [29], Switch Transformer [17], and Mixtral [26] are used to scale model parameters by tens or even hundreds of times while incurring far less than linear growth in computation. Distributed training of MoE models often spends roughly half of its execution time in all-to-allv communication [26, 33], and the communication volume changes dynamically with changes in expert routing decisions at runtime. Such MoE models are often trained on large-scale GPU-accelerated systems and push the underlying communication stack to its limits.

DL workloads, in particular, have significantly skewed distributions of data sizes exchanged between processes, dynamic changes in communication, and large message sizes. These make optimizing the all-to-allv communication challenging. When there is significant variation in the amount of data that different processes send and receive in a single all-to-allv operation, we refer to that as traffic imbalance or skew. When a few processes handle much more data than the rest, we say the communication is highly skewed. In this case, the completion time is dominated by the network interface card (NIC) used by the most imbalanced process: other NICs finish early and remain idle while the overloaded NIC drains its queue slowly, creating a long-tail effect. When communication traffic is lightly skewed, that is, when most processes send and receive similar amounts of data, no single process or NIC is a clear straggler. Instead, the bottleneck shifts to network-level congestion: all processes communicate with many peers simultaneously, and the resulting large number of concurrent flows contend for shared inter-node links, causing queueing delays, reduced effective bandwidth, and packet retransmissions [43].

In this paper, we propose a **S**kew-aware **A**ll-to-allv library for **B**alancing ir**RE**gular communication, SABRE[1], which optimizes large-scale all-to-allv operations under dynamically changing communication patterns on GPU-based systems. Our approach first measures the degree of communication volume imbalance using the maximum-to-mean (MTM) ratio. For highly-skewed all-to-allv, we derive and prove a theoretically optimal communication balancing scheme, then use high-bandwidth intra-node gathering and reordering to approximate this lower bound in practice while balancing traffic across processes within each node. For lightly-skewed communication patterns, we present a pipelined two-dimensional (2D) all-to-allv algorithm that overlaps intra-node and inter-node



ICS '26, Belfast, United Kingdom

ACM ISBN 979-8-4007-2522-7/26/07
https://doi.org/10.1145/3797905.3800541

[1] https://github.com/hpcgroup/sabre

communication to fully utilize intra-node bandwidth while minimizing idle inter-node links. In both regimes, we group inter-node communication to reduce network congestion.

We evaluate our algorithms on the Perlmutter supercomputer [32] at NERSC. Our results demonstrate significant improvements over both NCCL [1] and Cray MPICH in microbenchmarks and MoE training using Megatron-LM [39]. We package our implementation as a Python API that can directly replace `dist.all_to_all_single` in DL training workloads.

The key contributions of this work are as follows:

- We derive a theoretically optimal traffic balancing scheme for highly-skewed all-to-allv, and implement a practical approximation through intra-node gathering, inter-node grouping, and redistribution that balances communication traffic across processes within each node and approaches the theoretical lower bound.
- We propose a pipelined two-dimensional all-to-allv algorithm for lightly-skewed communication patterns that overlaps intra-node and inter-node communication to maximize bandwidth utilization while controlling network congestion.
- We design a skew-aware algorithm selection mechanism in SABRE that uses the MTM ratio to classify each all-to-allv invocation at runtime and dispatch it to the highly-skewed or lightly-skewed algorithm accordingly.
- We evaluate SABRE on the Perlmutter supercomputer, achieving up to 2.4× speedup over Cray MPICH and NCCL in microbenchmarks. We also demonstrate 1.28–1.79× end-to-end speedup for MoE models using Megatron-LM.

2 Background and Related Work

We provide a brief overview of mixture of experts models and all-to-allv communication patterns that arise in their distributed training. We then review static all-to-allv algorithms and recent scheduling-based optimization methods for modern GPU systems.

2.1 Mixture of Experts Models

Modern large language models are built on the Transformer architecture, which processes input tokens through a stack of layers, each consisting of a self-attention block and a feed-forward network (FFN). Scaling laws show that steadily increasing model size is an effective way to improve model quality. However, training dense models with trillions of parameters incurs computation and memory costs that are prohibitive for most machines. Mixture of Experts (MoE) architectures have emerged as a key technique to break this barrier. By introducing sparse activation, MoE models such as GShard, Switch Transformer, and Mixtral [17, 26, 29] can grow the number of parameters by tens or even hundreds of times while the increase in computation stays far below linear. The main idea is that for each input token, the model only activates a small subset of parameters (the "experts") instead of the full network.

As shown in Figure 1, MoE layers are typically used as drop-in replacements for the feed-forward networks (FFNs) in Transformer architectures, expanding model capacity through sparse computation. A typical MoE layer consists of two key components: a set of expert networks (each implemented as a small multilayer perceptron) and a gating network (also called a router).

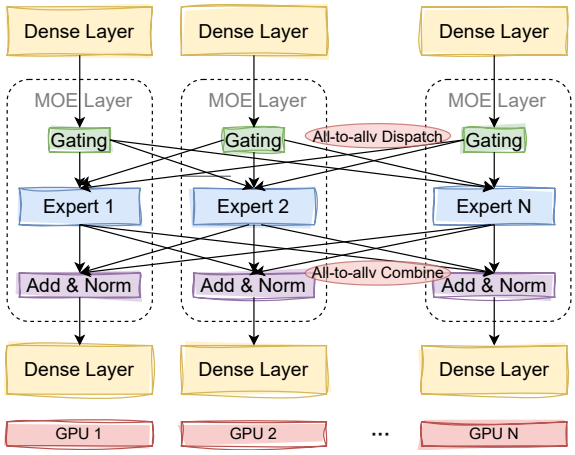


Figure 1: Expert parallelism in distributed training of a mixture of experts model with all-to-allv communication.

The gains from sparsity come with significant communication cost. When MoE models use expert parallelism, where different experts are placed on different GPUs, each layer introduces intensive all-to-allv communication. Figure 1 shows the all-to-allv pattern in MoE training with expert parallelism. Each MoE layer performs two all-to-allv collectives. The first dispatches tokens to remote GPUs that host the selected experts, and the second gathers the expert outputs back to the tokens' original GPUs. Prior work reports that all-to-allv can account for more than 50% of the total training time in large-scale MoE models [26, 33].

Optimizing MoE training has also drawn significant attention in recent years [19, 30]. Frameworks such as DeepSpeed-MoE [36, 40] focus on architectural optimizations. Others, including Tutel and Comet [25, 45], improve efficiency by overlapping computation with all-to-all communication. This paper is orthogonal to these approaches. We focus on optimizing all-to-allv, which is often the most expensive communication in MoE training.

2.2 Static All-to-allv Algorithms

In an all-to-all among P processes, process i sends a distinct data block to process j for all $j \in \{0, 1, \ldots, P-1\}$, while also receiving data from every other process. For most algorithms based on direct exchange, this pattern involves up to $P(P-1)$ point-to-point transfers, which makes all-to-all one of the most communication-intensive collectives. A large body of work focuses on optimizing all-to-all and all-to-allv communication in MPI [4, 8–10, 13–16, 23]. Below, we briefly review several widely used static algorithms.

Linear Algorithms: The fan out algorithm is a simple all-to-all implementation where all GPUs engage in simultaneous point-to-point communication with all peers, creating a large number of concurrent connections per process. This approach is used in RCCL [3] and in the default `dist.all_to_all_single` in PyTorch.

The spread out algorithm is another classic all-to-all implementation. It organizes communication into $P-1$ rounds. In round r, each rank exchanges data with a unique partner so that over all

rounds every rank pair communicates exactly once. This simple schedule keeps the number of active connections per process low and can reduce congestion compared with a naive fan out. Many MPI libraries, including MPICH [21] and OpenMPI [20], employ variants of the spread out algorithm because of its simplicity and its reasonable congestion behavior on homogeneous networks.

Another common approach is the pairwise exchange algorithm, which issues a nonblocking receive from one peer and uses a blocking send to that peer in each round. It then waits for the two requests to complete before moving on to the next partner. This strategy reduces the number of outstanding communications even further, but processes peers in a strictly sequential order.

Logarithmic Algorithms: The Bruck algorithm [5, 41, 42] optimizes all-to-all communication with logarithmic complexity. It executes $\log(P)$ exchange rounds, and in each round the communication distance doubles. In round k, process i exchanges data with process $(i \pm 2^k) \bmod P$. After each exchange, it performs local data reorganization to prepare messages for the next round. This recursive structure reduces the number of communication rounds from $O(P)$ to $O(\log P)$.

Several studies have optimized the Bruck algorithm and searched for optimal radix parameters for recursive constructions [13–15, 37]. These studies show that small radices work well for small messages, a radix close to $\sqrt{P}$ improves performance for mid-sized messages, and large radices benefit large messages in the kilobyte range [16]. The strength of Bruck style algorithms lies in latency-bound scenarios. Even though they increase the total data volume, the reduction in the number of communication rounds can lead to much lower latency and better performance. However, these algorithms mainly target small messages in latency-bound scenarios and do not address bandwidth-bound workloads.

Hierarchical Algorithms: Hierarchical all-to-allv algorithms target the layered architectures of modern HPC systems by decoupling the global exchange into intra-node and inter-node phases. Each node first aggregates data through high-bandwidth shared memory, then sends batched, coarse-grained messages across nodes. This approach reduces the number of concurrent cross-node messages and enables separate tuning for latency-sensitive intra-node and bandwidth-limited inter-node communication [13].

2.3 Scheduling-based Optimization Methods

Several recent works explore scheduling-based optimization methods for all-to-allv communication, especially for GPU-based deep learning workloads [7, 24, 28, 31, 38, 44]. These approaches model the full topology of GPU interconnects and the heterogeneous link bandwidths to generate near-optimal schedules. For example, SCCL [6] uses Satisfiability Modulo Theory solvers to synthesize collective algorithms. TACCL [38] formulates collective scheduling as a mixed integer linear programming problem, while TECCL [31] leverages traffic engineering techniques from multicommodity flow optimization to route messages efficiently across the network fabric. More recently, FAST [28] applies Birkhoff's theorem to decompose communication matrices into permutation matrices, and optimizes collectives on switch-based networks. Beyond hundreds of GPUs, however, these approaches require prohibitively long preprocessing. Some even take hours, making them impractical for deep learning training where communication patterns change dynamically.

3 Motivation and Scaling Analysis

In this section, we analyze the limitations of existing all-to-allv algorithms on modern GPU systems.

3.1 Bottlenecks in Static Algorithms

As discussed in Section 2.2, static algorithms such as fan out and spread out rely on fixed communication schedules for the all-to-allv operation. However, on GPU-based supercomputers these methods face two fundamental limitations. First, they assume a uniform network where bandwidth is the same between every pair of GPUs. In practice, GPU-based supercomputers have much higher intra-node bandwidth than inter-node bandwidth. A fixed schedule cannot exploit this gap, and thus fails to fully use the high-bandwidth intra-node interconnects. Second, their performance is limited by the rank that sends or receives the most data. In an all-to-allv operation with nonuniform message sizes, the busiest rank becomes a bottleneck. Other links stay idle while they wait for its transfers to finish, which wastes bandwidth and creates a long tail in completion time.

Radix-based Bruck variants try to tune schedules across different message sizes, but existing studies focus on messages up to the kilobyte scale. Deep learning training, however, uses message sizes from megabytes to gigabytes. In this regime, the extra traffic introduced by Bruck becomes counterproductive because the bottleneck shifts from latency to bandwidth, so Bruck algorithms fail to reach optimal performance for large messages.

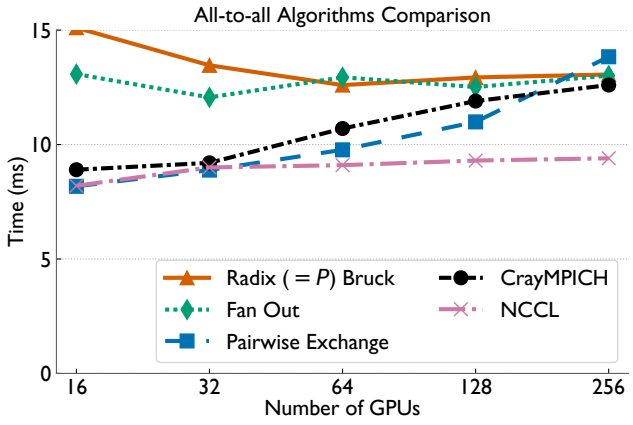


Figure 2: Performance of different all-to-all algorithms.

To understand how these algorithms behave on modern GPU systems in the large message regime, we benchmark their performance on Perlmutter. As shown in Figure 2, we compare several all-to-all algorithms, scaling from 16 to 256 GPUs with a 128 MB message. We use balanced all-to-all here to show that even in the best case where no skew exists, existing methods already hit fundamental performance limits on modern GPU systems. In practice, the highly-skewed patterns common in MoE training make these baselines perform even worse. For pairwise exchange and linear

(fan out) algorithms, we use the implementations provided by Open-MPI [20]. For the Bruck algorithm, we use the latest radix bruck implementation [13] and evaluate radix = P, which prior work reports as the best configurations for large messages [13]. When the radix equals P, radix Bruck degenerates to the linear case. Even with these choices, radix Bruck performs poorly in this bandwidth-bound regime. We also include implementations of Cray MPICH and NCCL. Overall, these results demonstrate that traditional static algorithms are no longer well suited to modern GPU systems in the large message regime. The NCCL-based fan-out implementation in PyTorch achieves the highest throughput. Research [22] indicates that NCCL automatically optimizes communication within an NCCL group. For instance, it reduces overhead by aggregating launches and achieves data transfer parallelism through multichannel allocation. In addition, NCCL is system- and topology-aware, which allows it to better exploit intra-node bandwidth. We believe these optimizations together help explain the strong performance of NCCL-based fan-out at scale.

3.2 Bottlenecks in Scheduling Algorithms

The scheduling based methods reviewed in Section 2 model the full GPU cluster topology and can produce schedules with near-optimal makespan for a fixed communication matrix. However, the cost of computing the schedule becomes prohibitive. For example, TACCL's scheduler can take over an hour for a 64 GPU cluster. FAST reduces complexity relative to mixed integer programming approaches, but its reported $O(n^5)$ runtime is still impractical for large deployments. A single all-to-all transfer typically finishes in milliseconds, so schedule computation that takes seconds only makes sense if it can be amortized across many identical transfers. In real deep learning training, routing decisions change across iterations and the communication matrix shifts accordingly, leaving little opportunity to reuse a precomputed schedule.

These findings motivate an all-to-allv optimization algorithm that addresses multi-level network topologies and GPU-centered architectures, adapts to dynamic large message communication patterns, and keeps runtime overhead low enough for practical deep learning workloads.

4 Overview of the SABRE Library

We first provide an overview of our SABRE library and design considerations for the skew-aware all-to-allv algorithms. We target production GPU-based supercomputers, where compute nodes have G GPUs and m NICs with $G = m$. We bind each process to a fixed NIC on every node, establishing a "1:1" mapping between NICs and processes. However, the framework generalizes naturally to systems where $G > m$ by defining communication groups as $\mathcal{G}_i = \{p \mid p \bmod m = i\}$ where p is the global rank, so that each group contains G/m GPUs sharing a single NIC. The theoretical lower bound derived in Section 5.2 still holds, and the load-balancing granularity shifts from per-GPU to per-NIC accordingly.

Design goals: Our goal is to build an adaptive library that chooses different optimization strategies based on how skewed the communication is. We aim to minimize inter-node congestion while keeping inter-node bandwidth as close to fully utilized as possible. This design is guided by the observation that all-to-allv bottlenecks usually stem from limited inter-node bandwidth. For example, each node has only four or eight GPUs but needs to exchange data with all other nodes, so the total inter-node communication volume far exceeds intra-node communication. As a result, when communication is highly-skewed, the completion time of an all-to-allv operation is dominated by the process that sends or receives the largest amount of data. When communication is relatively balanced, inter-node congestion becomes the main performance limitation.

Skewness metric: To quantify communication imbalance, we need a metric that captures how imbalanced the busiest process is compared to the average case. Traditional dispersion measures such as the coefficient of variation are not a good fit here. This is because we care less about overall randomness and more about the process that handles the largest cross-node communication. The largest communication volume across all processes is therefore a better predictor of all-to-allv completion time. To make the metric comparable across different message sizes, we normalize by the mean communication volume per process, so that the same threshold can drive algorithm selection on different problem sizes.

We formalize this maximum-to-mean (MTM) ratio as follows. For each rank i, let s_i denote the total inter-node data volume it sends and r_i denote the total inter-node data volume it receives. We define two MTM ratios,

$$\text{Send-MTM} = \frac{\max_i s_i}{\frac{1}{P}\sum_{j=1}^{P} s_j} = \frac{P \cdot \max_i s_i}{\sum_{j=1}^{P} s_j}$$

$$\text{Recv-MTM} = \frac{\max_i r_i}{\frac{1}{P}\sum_{j=1}^{P} r_j} = \frac{P \cdot \max_i r_i}{\sum_{j=1}^{P} r_j}$$

where P is the total number of processes. The overall MTM is,

$$\text{MTM} = \max(\text{Send-MTM}, \text{Recv-MTM}).$$

Higher MTM ratio indicates more severe skew. At runtime, we compute MTM ratio using the communication matrix and use it as a single scalar feature to decide whether to treat the current all-to-allv as highly-skewed or lightly-skewed.

4.1 Library Design

As illustrated in Figure 3, we design a skew-aware adaptive all-to-allv library (SABRE) for balancing irregular communication. We first compute the degree of imbalance using the communication matrix, and then we use the algorithm selector to choose the best all-to-allv algorithm based on this metric.

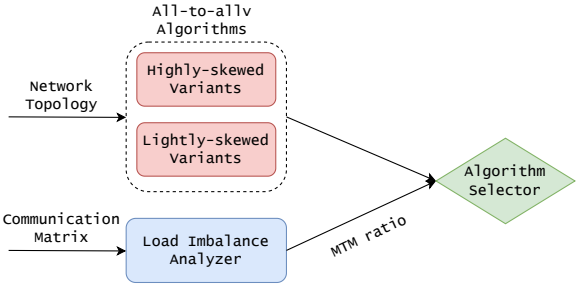


Figure 3: Components of the SABRE library

When MTM ratio is high, which means communication is highly-skewed, we first derive a theoretically optimal scheme that evenly balances each node's total send and receive communication across its NICs. We then use high bandwidth intra-node gathering and reordering to approximate this lower bound while avoiding unnecessary fragmentation. We aggressively partition very large data blocks and distribute them across multiple NICs, which prevents oversubscription of individual inter-node links without exploding the number of messages. For inter-node communication, we batch and coalesce chunks so that each pair of nodes exchanges a small number of large, well-balanced transfers.

When MTM ratio is low and traffic is lightly-skewed, we switch to a pipelined 2D all-to-allv algorithm that overlaps intra-node and inter-node phases. In this regime, the main goal is to control inter-node congestion rather than to fix imbalance. We use the abundant intra-node bandwidth to relay data within each node while simultaneously grouping inter-node communication into batched exchanges, which reduces the number of inter-node messages and alleviates pressure on the network fabric. In the next two sections, we describe the detailed algorithms and implementation for both highly-skewed and lightly-skewed regimes.

5 Algorithm for Highly-skewed All-to-allv

We present a derivation of the theoretically optimal form for highly-skewed all-to-allv communication and then an implementation that closely approximates this lower bound in practice.

5.1 Problem Definition

We focus on GPU-based clusters that provide much higher intra-node bandwidth than inter-node bandwidth. Under this condition, we represent the system as a 2D grid of processes with dimensions $N \times G$, where N is the number of nodes and G is the number of GPUs per node. The total number of processes is $P = N \times G$.

Each node has m NICs, and each NIC has inter-node bandwidth C. We focus on inter-node transfers, since intra-node bandwidth is much larger than C and does not limit performance. We use $v_{u\to w}$ to denote the total data volume that node u sends to node w. The goal is to minimize completion time T of the all-to-allv.

Note that the inter-node bandwidth between any pair of nodes is the same. Under this assumption, forwarding messages through intermediate nodes cannot reduce the all-to-allv completion time, as each byte still uses the same bottleneck bandwidth. In the rest of this section, we therefore focus on schedules that use only direct inter-node transfers, without loss of optimality.

5.2 Theoretical Optimality Analysis

In a highly-skewed all-to-allv, a few NICs carry most of the inter-node traffic while the rest remain idle. The most loaded NIC therefore dictates the completion time. If, instead, each node spreads its total inter-node send and receive data evenly across all m NICs, every NIC finishes at roughly the same time and no capacity goes to waste. We prove that this equal-spreading strategy is not merely a heuristic but achieves the tightest possible completion time under the inter-node bandwidth constraints, making it globally optimal.

Let node u have total send volume $S_u = \sum_w v_{u\to w}$, and let node w have total receive volume $R_w = \sum_u v_{u\to w}$. We use $x_{u\to w}^{(a,b)} \geq 0$ to denote the data sent from node u's ath NIC to node w's bth NIC. The total data across all NIC pairs between two nodes equals the total volume $v_{u\to w}$:

$$\sum_{a=1}^{m}\sum_{b=1}^{m} x_{u\to w}^{(a,b)} = v_{u\to w}.$$

Each NIC can transmit at most $C \cdot T$ bytes in time T, which gives the per-port capacity constraints:

$$\sum_{w}\sum_{b} x_{u\to w}^{(a,b)} \leq CT; \quad \sum_{u}\sum_{a} x_{u\to w}^{(a,b)} \leq CT; \quad \forall u, w, a, b \in [1, m].$$

The first constraint limits the total data any single NIC can send to all destinations; the second limits how much any single NIC can receive. Our objective is to minimize T. Summing the send constraint over all m ports of node u gives

$$\sum_{a=1}^{m}\sum_{w}\sum_{b} x_{u\to w}^{(a,b)} \leq mCT \Rightarrow S_u \leq mCT \Rightarrow T \geq \frac{S_u}{mC}, \quad \forall u.$$

This means the completion time is at least the time needed for the heaviest-sending node to push all its data through its m NICs. Similarly, for all receivers w:

$$T \geq \frac{R_w}{mC}, \quad \forall w.$$

Combining both directions, we obtain the lower bound T_{LB}:

$$\boxed{T \geq T_{\mathrm{LB}} := \max\Big(\max_u \frac{S_u}{mC},\ \max_w \frac{R_w}{mC}\Big).}$$

In words, the all-to-allv completion time is bounded below by the node with the largest total send or receive volume, since that node must drain all its data through its m NICs.

We now give a constructive proof that if each node evenly distributes its total send and receive data across its m inter-node ports, then there exists a feasible schedule that achieves $T = T_{\mathrm{LB}}$, and is therefore globally optimal. The key idea is to split the data volume $v_{u\to w}$ uniformly across all NIC pairs between nodes u and w. For each pair (u, w), we define an $m \times m$ doubly stochastic matrix $Q^{(u,w)} = [q_{a,b}^{(u,w)}]$ that describes how to distribute $v_{u\to w}$ across NIC pairs:

$$\sum_{b=1}^{m} q_{a,b}^{(u,w)} = \frac{1}{m}, \quad \sum_{a=1}^{m} q_{a,b}^{(u,w)} = \frac{1}{m}, \quad q_{a,b}^{(u,w)} \geq 0.$$

These constraints ensure that each sending NIC handles exactly $1/m$ of the total volume, and each receiving NIC also handles exactly $1/m$. We then set

$$x_{u\to w}^{(a,b)} = v_{u\to w}\, q_{a,b}^{(u,w)}.$$

Under this assignment, each sending NIC (u, a) carries exactly $1/m$ of node u's total send volume:

$$\sum_{w}\sum_{b} x_{u\to w}^{(a,b)} = \sum_{w} v_{u\to w}\left(\sum_{b} q_{a,b}^{(u,w)}\right) = \sum_{w} v_{u\to w} \cdot \frac{1}{m} = \frac{S_u}{m}.$$

Similarly, each receiving NIC (w, b) carries exactly $1/m$ of node w's total receive volume:

$$\sum_{u}\sum_{a} x_{u\to w}^{(a,b)} = \sum_{u} v_{u\to w}\left(\sum_{a} q_{a,b}^{(u,w)}\right) = \sum_{u} v_{u\to w} \cdot \frac{1}{m} = \frac{R_w}{m}.$$

Since no NIC sends or receives more than $\max(S_u, R_w)/m$ data, all port constraints are satisfied when

$$T \geq \max\Big(\max_u \frac{S_u/m}{C},\ \max_w \frac{R_w/m}{C}\Big) = T_{\text{LB}},$$

so the schedule is feasible at $T = T_{\text{LB}}$. Combined with the lower bound above, we obtain

$$\boxed{T^\star = T_{\text{LB}}.}$$

Therefore, under the inter-node bandwidth bottleneck, and direct inter-node transfers only, we can achieve global optimality by evenly distributing each node's inter-node communication across its m NICs. In other words, to minimize the completion time of an all-to-allv operation in highly-skewed scenarios, each node should balance its total send and receive loads evenly across all available NICs. This strategy mitigates the impact of communication imbalance and improves overall communication performance.

5.3 Three-phase All-to-allv Algorithm

We now present a practical implementation on Perlmutter that approximates this theoretically optimal strategy on real systems. Figure 4 gives an overview of our highly-skewed 2D all-to-allv design. It highlights three phases: intra-node gathering and load balancing, inter-node communication, and intra-node distribution with final assembly. These phases work together to produce an implementation that closely matches the theoretical optimum derived above. For clarity, the figure only shows data transfers between two nodes; larger deployments follow the same pattern.

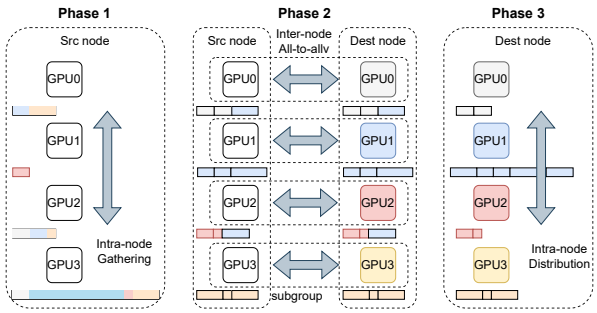


Figure 4: Highly-skewed 2D all-to-allv algorithm.

5.3.1 Communication Group Partitioning. Each process has a global rank $p = n \times G + g$, where $n \in [0, N-1]$ is the node index and $g \in [0, G-1]$ is the local GPU index. We define the i-th communication group as:

$$\mathcal{G}_i = \{p \mid p \bmod m = i\}, \quad i = 0, 1, \ldots, m-1,$$

which means we partition all processes into m inter-node communication groups based on rank mod m. Each group is responsible for carrying roughly $1/m$ of the node's inter-node data. This partitioning strategy provides two key benefits:

Receive side balancing. Our optimality analysis shows that all-to-allv performance is highest when each node spreads its inter-node communication evenly across its m NICs. In our design, as shown in Algorithm 1 (line 3), load balancing is applied separately to each destination node. For a fixed source node u and destination node w, if the volume $v_{u\to w}$ is evenly split across the m communication groups on node u, then node w receives the same total volume of data in each group as well. As a result, our algorithm only needs to optimize the send side to be balanced, and the receive side balancing is automatically achieved by this group partitioning.

Reduced NIC concurrency. Group based inter-node communication also reduces the number of concurrent messages on each NIC. Without partitioning, a single NIC may need to handle messages from all $N \times G$ processes at once. With partitioning, each NIC only serves processes in its own group, roughly $\frac{N\times G}{m}$ processes. This strategy reduces contention and short-term queueing at the NIC level. In effect, it turns many small communications into fewer but larger ones, which improves bandwidth utilization.

Algorithm 1 Intra-node Greedy Traffic Balancing and Splitting

Require: Communication matrix M, source node N_S, intra-node GPUs $G = \{g_0, \ldots, g_{G-1}\}$
Ensure: Assignment map A: $g_{proxy} \to$ list of data blocks

1: $\mathcal{B}_{\text{dest}} \leftarrow$ _collect_node_data_blocks(M, N_S, G) ▷ Group blocks by destination node
2: $A \leftarrow$ empty map, $A[g_i] \leftarrow \emptyset$ for $i \in [0, G-1]$
3: **for** each destination node N_D in $\mathcal{B}_{\text{dest}}$ **do**
4: $P_D \leftarrow \text{SortBySize}(\mathcal{B}_{\text{dest}}[N_D])$ ▷ Sort blocks by size
5: $L_{\text{avg}}, L_{\max} \leftarrow \text{InitSubproblem}(P_D, G)$
6: $\mathbf{L}_{\text{proxy}} \leftarrow$ array of size G initialized to 0
7: **while** P_D is not empty **do**
8: $b \leftarrow P_D$.pop_front()
9: $g_{pref} \leftarrow G[b.dest\_gpu\_id]$
10: $g_{min} \leftarrow \text{argmin}_{g\in G}(\mathbf{L}_{\text{proxy}}[g])$
11: $(g_{target}, need\_split) \leftarrow \text{SelectTarget}(b, g_{pref}, g_{min})$
12: **if** *need_split* **then**
13: $L_{avail} \leftarrow L_{\max} - \mathbf{L}_{\text{proxy}}[g_{target}]$
14: $b_1, b_2 \leftarrow \text{split\_block}(b, L_{avail})$
15: $A[g_{target}]$.append(b_1)
16: $\mathbf{L}_{\text{proxy}}[g_{target}] \leftarrow \mathbf{L}_{\text{proxy}}[g_{target}] + b_1.size$
17: P_D.insert_front(b_2) ▷ Return remainder to queue
18: **else**
19: $A[g_{target}]$.append(b)
20: $\mathbf{L}_{\text{proxy}}[g_{target}] \leftarrow \mathbf{L}_{\text{proxy}}[g_{target}] + b.size$
21: **end if**
22: **end while**
23: **end for**
24: **return** A

5.3.2 Intra-node Gathering and Traffic Balancing. In the data preparation phase, the algorithm iterates over all local GPUs to collect data blocks destined for other nodes (line 1). These blocks are grouped by their destination node, which splits the work into multiple independent subproblems, one for each destination node (line 3). For the subproblem corresponding to a specific destination node N_D, the algorithm sorts the data blocks by size in decreasing order (line 4) and computes the total data volume L_{total} and the ideal average proxy load $L_{avg} = L_{total}/G$ (line 5), where G is the number of processes on the node. On Perlmutter, we have $G = m$, so

balancing per-GPU load is equivalent to balancing NIC load within a node. This matches the assumptions in our optimality analysis.

The next step is to distribute these data blocks among the local GPUs. A naive attempt to achieve perfect load balance can create many tiny blocks, which adds overhead without much benefit. To avoid this, we introduce a load tolerance threshold $L_{max} = \alpha \cdot L_{avg}$, which allows a small amount of load imbalance and prevents harmful over-splitting. In our sensitivity tests, values of α from 1.02 to 1.08 yield similar performance across all evaluated scales and message sizes. We empirically set $\alpha = 1.05$ as a robust default.

When processing a data block destined for GPU g_{dest} on node N_D (line 8), the algorithm first tries to assign it to the GPU g_{proxy} on the source node that has the same local ID (that is, $g_{proxy}.id = g_{dest}.id$, line 9). The key advantage of this design is that it simplifies data handling on the destination node. When the data block arrives at N_D, it already resides on the correct target GPU, so the final distribution phase does not need additional intra-node forwarding. This significantly reduces communication pressure on the destination node. If the preferred GPU is full (that is, receiving the block would exceed L_{max}), the algorithm triggers a fallback mechanism and assigns the block to the least loaded GPU g_{min_load} on the source node (lines 10–11). If the block is still too large for g_{min_load}, the algorithm performs dynamic splitting (lines 13–14). Then the chosen GPU accepts the largest possible chunk b_1 that fills its capacity up to L_{max} (lines 15–16), and the remaining portion b_2 is pushed back to the front of the pending queue (line 17). This ensures that the remainder is handled quickly by the next available GPU. The greedy process repeats (line 7) until all data blocks are assigned.

After the assignment step, the source node performs an intra-node shuffle in which the original source GPUs send their data blocks to the assigned proxies. This turns the dense GPU-to-GPU pattern into a small number of streams on each node.

5.3.3 Inter-node Communication. In the inter-node transfer phase, each communication group sends data in large batched transfers between nodes. Because intra-node gathering has already merged per GPU streams, the remaining communication behaves as an $N \times N$ node-level exchange rather than a dense $P \times P$ GPU-level exchange, where P is the world size and N is the number of nodes. These larger node-to-node batches reduce NIC concurrency and short-term queueing. Moreover, for point-to-point communication, the receiver must know the incoming message size before it can post the receive. To avoid an extra metadata exchange, we run the same splitting algorithm on the sending and receiving sides. Given the shared communication matrix, both sides independently compute the same message layout and sizes, so receivers can prepare buffers and issue P2P receives without any metadata exchange.

5.3.4 Intra-node Distribution and Final Assembly. After the inter-node transfer finishes, each GPU holds a set of data fragments received from different remote nodes, and these fragments are destined for different final GPUs on the same node. The receiving side then performs two steps: intra-node distribution and final assembly. First, it runs an intra-node all-to-allv according to a precomputed distribution plan, which forwards each fragment to its final target GPU. Second, each target GPU builds its slice of the output tensor by writing two types of data: 1) resident fragments that already sit on the correct GPU, a direct result of our ID matching heuristic, and 2) forwarded fragments that arrive from other local GPUs through the intra-node all-to-allv.

6 Algorithm for Lightly-skewed All-to-allv

In many deep learning workloads, not every all-to-allv operation suffers from severe communication imbalance. For example, auxiliary loss terms in MoE training encourage a more uniform token distribution across experts as training progresses. As training goes on, the all-to-allv communication therefore becomes much more balanced. When the MTM ratio is relatively low, continuing to run the highly-skewed traffic balancing algorithm only adds overhead. The traffic balancing optimization in Section 5.3 relies on extra forwarding and reassembly steps, which are justified only when a few GPUs are much more imbalanced than the rest. Thus, for lightly-skewed cases, we design a 2D algorithm that focuses on congestion control rather than explicit communication balancing.

6.1 Problem Definition

In the lightly-skewed regime, there are no clear stragglers or long-tail effects. The total data sent and received by each process is already close to balanced. The bottleneck shifts from traffic imbalance to congestion, where all processes communicate with many peers at the same time and create strong contention in the inter-node network.

Note that inter-node communication dominates intra-node communication in all-to-allv. Consider a system with N nodes and G GPUs per node, for a total of $P = N \times G$ processes. Each process must communicate with the other $P - 1$ processes. Under a uniform communication matrix, each process forwards and receives roughly $\frac{(N-1)\times G}{N\times G}$ of its data through inter-node links, while only a fraction of $\frac{G-1}{N\times G}$ uses intra-node links. For example, with $N = 16$ nodes and $G = 4$ GPUs per node, each process sends about 93.75% of its data over inter-node links, 4.68% over intra-node links, and the remaining 1.56% stays local. A naive fan out implementation of all-to-allv therefore leaves most of the high-bandwidth intra-node interconnects underutilized.

Therefore, in the lightly-skewed all-to-allv regime, our optimization strategy is to use the abundant intra-node bandwidth to relay and gather data, thereby reducing contention and congestion on the inter-node network.

6.2 Two-phase All-to-allv Algorithm

As discussed in Section 5, grouping communication is an effective way to reduce the number of concurrent transfers and ease congestion. We adopt a similar idea here but adapt it to the lightly-skewed case. A simple design would perform all intra-node forwarding first and then start inter-node transfers, which we refer to as the *SABRE w/o overlap* algorithm. However, this would serialize the two phases and create pipeline stalls.

We observe that each process's send buffer contains three kinds of data:

(1) data for intra-node all-to-allv (destined for GPUs within the same node);
(2) data that can be exchanged directly within the current inter-node communication groups via all-to-allv;

(3) data that should be forwarded to a proxy GPU on the same node for transmission to other inter-node groups.

To exploit this structure and avoid stalls, we divide all-to-allv communication into two overlapping phases, as illustrated in Figure 5. For clarity, Figure 5 shows part of the all-to-allv communication; the pattern for the other processes is similar.

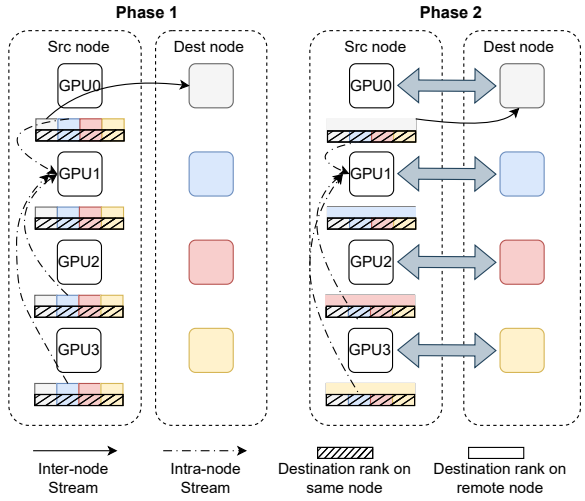


Figure 5: Lightly-skewed 2D all-to-allv algorithm. For clarity, only part of the communication is shown; the pattern for the other processes is similar.

Phase 1: direct exchange with forwarding reception. Each GPU performs all-to-allv operations with its assigned inter-node communication groups, sending and receiving data destined for that group. At the same time, each GPU acts as a proxy and receives data from other GPUs on the same node that are destined for that inter group. This phase uses both inter-node links (for the direct exchange) and intra-node links (for receiving forwarded data).

Phase 2: proxy forwarding with intra-node processing. Each GPU forwards the data it received from other local GPUs in Phase 1 to the appropriate destinations within its inter-node communication groups. At the same time, GPUs can process data that needs to be distributed within the node via intra-node all-to-allv. This phase again uses both inter-node links (for forwarding) and intra-node links (for local distribution).

6.3 Parallelism and Performance Benefits

A key advantage of this two-phase design is that the operations in each phase can run in parallel because they use different resources. Intra-node and inter-node links operate independently. The algorithm keeps both kinds of links busy and uses the abundant intra-node bandwidth to relieve pressure on the inter-node network. In Phase 1, the amount of data forwarded within a node is larger than the data sent across nodes. However, intra-node links provide much higher bandwidth than inter-node links, so local forwarding and the cross-node exchange finish in roughly the same time. In other words, we almost get message gathering for free.

Compared with a naive point-to-point based implementation, our lightly-skewed 2D all-to-allv provides three main benefits:

- **Reduced congestion:** grouped communication reduces the number of concurrent inter-node connections from $O(P^2)$ to $O(N^2)$ and eases network congestion.
- **Improved link utilization:** high-bandwidth intra-node links are used for both forwarding and local distribution, instead of sitting idle.
- **Pipeline efficiency:** overlapping direct exchange with forwarding avoids pipeline stalls and keeps throughput high.

7 Implementation Details

In the previous sections, we introduced our all-to-allv methods for both highly-skewed and lightly-skewed communication patterns. This section explains how we implement these methods on the Perlmutter supercomputer, including algorithm selection and communication backend selection. We implement the algorithms in C++ and provide pybind11 bindings so they can act as a direct replacement for `dist.all_to_all_single`, which is widely used in deep learning training workloads.

7.1 Imbalance Detection & Algorithm Selection

To choose the appropriate all-to-allv algorithm for a given pattern, we first need to measure how imbalanced the communication matrix is. In deep learning frameworks such as Megatron-LM, each process already computes the full all-to-allv metadata (per-destination send counts) as part of the MoE routing and token dispatch logic. SABRE directly reuses this existing metadata, and therefore does not require any additional global exchange. For applications that do not provide global metadata, SABRE only requires an intra-node allgather of the per-rank send and receive size vectors, which are typically a few KB and negligible compared with the 64 MB–512 MB data transfers in our target workloads. The MTM ratio computation and algorithm planning are performed locally, and their overhead is included in all end-to-end measurements reported in Section 8.4.

We use the maximum-to-mean ratio (MTM) defined in Section 4 as our imbalance metric. $MTM = 1$ corresponds to a perfectly balanced communication matrix in which every GPU sends and receives the same amount of data, while a larger MTM ratio indicates that some GPUs handle more data. For example, $MTM = 8$ means that one GPU handles 8× the average load in an all-to-allv operation, which represents a severely imbalanced pattern.

Figure 6 shows how all-to-allv completion time changes with MTM ratio for both the highly-skewed and the lightly-skewed algorithm. In this figure, we report results for average message sizes of 64 MB and 128 MB. For both message sizes, the crossover point where the highly-skewed algorithm becomes faster than the lightly-skewed algorithm stays near $MTM \approx 2.2$. As discussed in Section 4, we normalize MTM ratio by the mean data volume so that the metric remains comparable across different message sizes. This normalization allows us to use a single MTM ratio threshold to choose between the two algorithms at a fixed GPU count even when the average message size changes. When MTM ratio is below roughly 2.2, the lightly-skewed algorithm achieves up to 15% lower latency than the highly-skewed algorithm. In this lightly-skewed regime, trying to fully balance the load would add

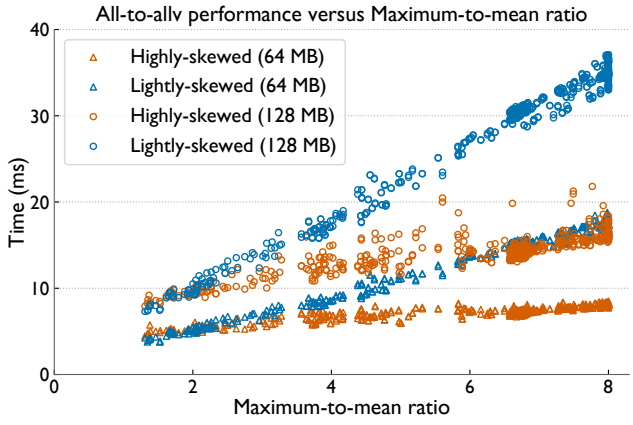


Figure 6: All-to-allv completion time versus maximum-to-mean ratio (MTM) on 16 GPUs for the highly-skewed and lightly-skewed algorithms.

forwarding overhead without enough benefit. However, once MTM ratio exceeds about 2.2, the lightly-skewed algorithm can no longer fully utilize the available inter-node bandwidth. Congestion control alone is not enough to offset the impact of the hottest GPUs. In this regime, the highly-skewed algorithm that redistributes data across processes becomes necessary and provides up to 2.3× speedup over the lightly-skewed algorithm when MTM ratio closes to 8. We also run similar tests for other GPU counts and choose the algorithm based on these results.

7.2 Communication Backend Selection

Modern GPU clusters typically have heterogeneous network connections. To achieve good performance on such systems, we need to pick communication backends for both intra-node and inter-node transfers. In this paper, both the highly-skewed and lightly-skewed algorithms rely on intra-node communication to forward messages between GPUs. This forwarding step is essentially an intra-node all-to-all operation. To choose the intra-node backend, we benchmark the intra-node all-to-all using NCCL and MPI for message sizes of 64, 128, 256, and 512 MB.

As shown in Table 1, the NCCL-based all-to-all implementation clearly outperforms MPI and delivers up to 2× higher bandwidth for large message sizes. Additionally, prior work has shown that NCCL is heavily optimized for intra-node GPU communication [12]. These results lead us to use NCCL for all intra-node data exchanges.

Table 1: Intra-node all-to-all performance comparison between NCCL and MPI

Message size (MB)	MPI time (ms)	NCCL time (ms)
64	0.726	0.329
128	1.326	0.588
256	2.513	1.079
512	4.887	2.077

We observe a similar trend for inter-node communication. When we fix NCCL as the intra-node backend and vary only the inter-node backend, NCCL again matches or exceeds the performance of MPI (Cray MPICH) across all scales. Guided by this finding, SABRE relies on NCCL for both intra-node and inter-node GPU communication. Note that our NCCL baseline (`dist.all_to_all_single` in PyTorch) implements all-to-allv as P pairs of `ncclSend`/`ncclRecv` calls inside a single `ncclGroupStart`/`ncclGroupEnd` block, which is the standard approach in major DL training frameworks.

8 Performance Results

We now present a comprehensive evaluation of our all-to-allv library on a production GPU-based supercomputer. We use realistic communication patterns derived from MoE training jobs.

8.1 Experimental Setup

All experiments are run on the Perlmutter [32] supercomputer at NERSC. Each GPU node has one 64-core AMD EPYC 7763 Milan CPU with 256 GB of DDR4 memory and four NVIDIA A100 GPUs. Within each node, the four GPUs are fully connected by third-generation NVLink [34], providing 100 GB/s of bandwidth per GPU pair. Across nodes, Perlmutter uses the HPE Slingshot 11 [11] interconnect in a three-hop dragonfly topology [27]. Each node has four HPE Cray Cassini [2] NICs, each providing 25 GB/s, for an aggregate inter-node injection bandwidth of 100 GB/s per node.

We compare SABRE with two widely used communication libraries: Cray MPICH and NCCL. We use CUDA 12.9 for GPU programming, NCCL 2.27.3 for GPU-based collectives, and Cray MPICH 8.1.30 with libfabric 1.22.0 for MPI-based communication. For each combination of library, message size, and GPU count, we run 10 independent trials and report the average completion time.

To keep the evaluation realistic, we extract all-to-allv communication matrices from MoE training jobs. MoE models produce dynamic and highly imbalanced communication patterns, which make them a good stress test for all-to-allv implementations.

Highly-skewed communication pattern. This workload represents the early phase of MoE training, when most tokens are routed to a small subset of experts due to initial model bias. For example, with 16 experts, expert parallelism degree 16, and top $k = 2$ routing, nearly all data is directed to only two GPUs.

Lightly-skewed communication pattern. This workload represents later training stages, typically after the auxiliary load-balancing loss has taken effect. At this point, communication is more evenly spread across experts, though some skew remains. The MTM ratio in this scenario fluctuates between 1 and 2.5.

8.2 Performance of Highly-skewed All-to-allv

We first study the highly-skewed communication pattern. Figure 7 reports the completion time of Cray MPICH, NCCL, and SABRE for buffer sizes of 128 MB and 256 MB as we scale the job from 16 to 256 GPUs. Cray MPICH and NCCL have almost the same completion time in all cases. Both baselines lack explicit traffic balancing, so the slowest processes determine performance, confirming that inter-node bandwidth is the main bottleneck.

In contrast, SABRE delivers much lower completion time across all settings. As discussed in Section 5, SABRE first performs greedy communication gathering and balancing inside each node, then groups inter-node transfers into a small number of balanced flows, and finally redistributes data within the destination nodes. This

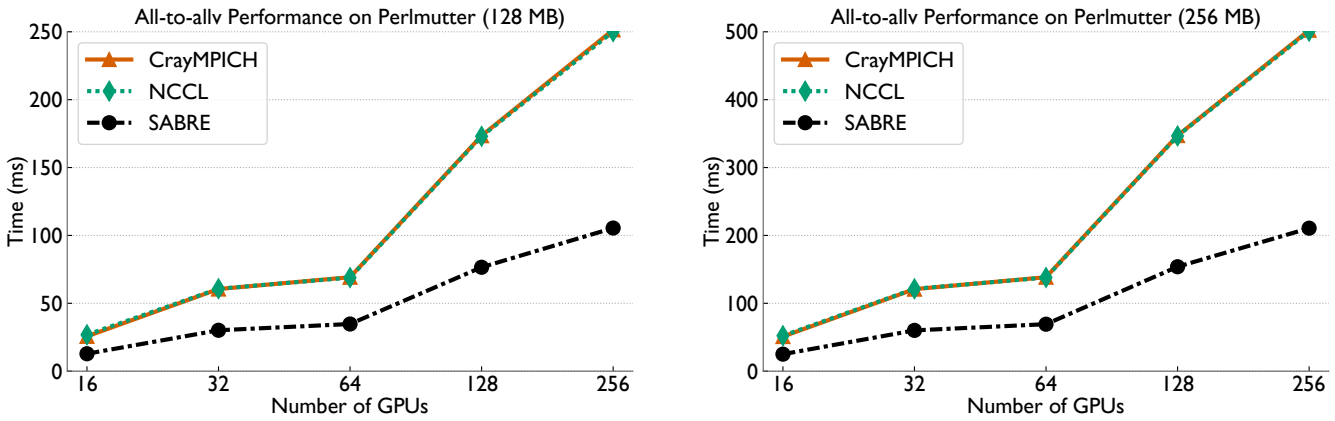


Figure 7: All-to-allv performance under highly-skewed communication patterns, where message size denotes the average data volume sent per process. SABRE consistently outperforms Cray MPICH and NCCL across GPU counts and message sizes.

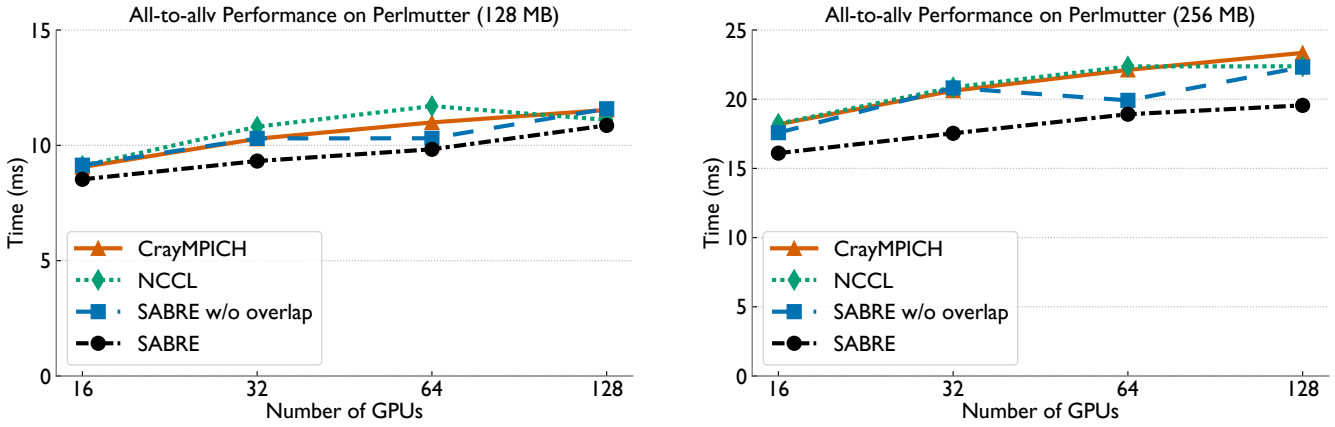


Figure 8: All-to-allv performance under lightly-skewed communication patterns, where message size denotes the average data volume sent per process. SABRE consistently outperforms Cray MPICH and NCCL across GPU counts and message sizes.

design evens out send and receive data on every node and removes stragglers. As a result, SABRE achieves up to 1.8×, 1.9×, 2.3×, 2.4×, and 2.4× speedup over the best baseline at 16, 32, 64, 128, and 256 GPUs, respectively.

Moreover, Figure 9 (left) further shows SABRE's speedup over NCCL in the highly-skewed setting for various message sizes and GPU counts. These results indicate that SABRE consistently provides significant speedups across all message sizes and GPU counts.

8.3 Performance of Lightly-skewed All-to-allv

We next examine the lightly-skewed communication pattern, which represents later training stages where the auxiliary loss has already made routing more balanced. Figure 8 shows the results on Perlmutter for 16, 32, 64, and 128 GPUs and uses buffer sizes of 128 MB and 256 MB. Under this workload, the gap between Cray MPICH and NCCL remains small, since both are limited by inter-node congestion. As described in Section 6.2, the SABRE w/o overlap baseline only partitions the system into a 2D mesh. Each GPU exchanges data with peers that are in the same inter-node group (`rank % GPUs_per_node`) and forwards residual communication, but it does not overlap intra-node and inter-node communication, so pipeline stalls remain. As shown in Figure 8, SABRE w/o overlap still has better performance than NCCL and Cray MPICH, which shows that this 2D grouping effectively mitigates congestion.

SABRE continues to outperform the baselines. Across all GPU counts and message sizes, SABRE achieves the lowest completion time and runs 10–30% faster than Cray MPICH and NCCL. The 2D all-to-allv design for the lightly-skewed regime overlaps intra-node forwarding with inter-node exchange, actively uses fast intra-node links to relieve pressure on the network. This overlap keeps intra-node links busy, reduces contention on inter-node links, and yields consistent speedup across all GPU counts and message sizes. The contrast between SABRE w/o overlap and SABRE highlights that pipelining and overlap optimizations are necessary to fully exploit the all-to-allv operation. Figure 9 (right) shows speedups over NCCL. SABRE maintains strong speedups for all large message sizes in the lightly-skewed case. However, for very large GPU counts with medium total message sizes per process, SABRE can be slower than

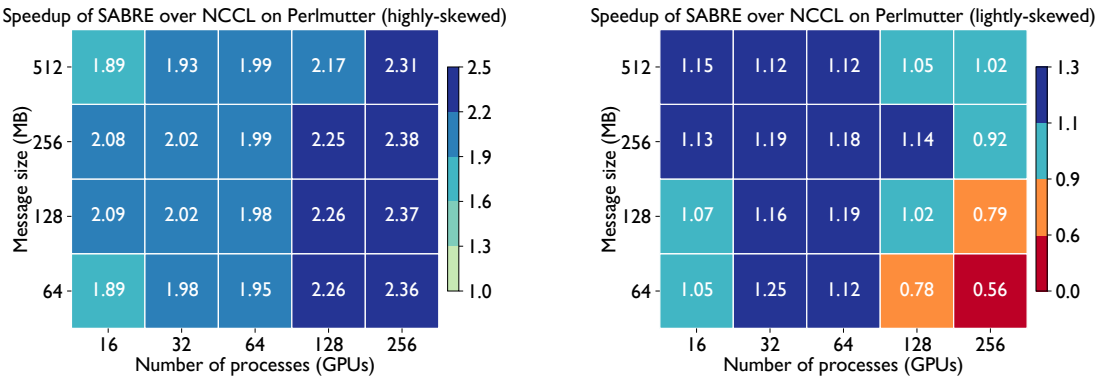


Figure 9: Heatmaps showing speedups from using SABRE over NCCL for highly-skewed (left) and lightly-skewed (right) all-to-allv on Perlmutter. Message size denotes the average data volume sent per process.

NCCL. This behavior is expected. In this paper, the message size is the total amount of data each process sends. When we increase the number of GPUs while keeping this total message size fixed, the average block size per peer becomes very small. For example, with a 64 MB message size and 256 GPUs, each block sent to a peer is only about 64 MB/256 ≈ 256 KB. In this regime, the workload falls outside the sweet spot of our pipelined 2D all-to-allv algorithm.

8.4 Impact on Application Performance

To understand how our all-to-allv optimizations behave in real applications, we evaluate SABRE in an end-to-end MoE training workload. We integrate SABRE into Megatron-LM [39], a widely used framework for large-scale transformer models and MoE training. The baseline uses the default `dist.all_to_all_single` kernel in PyTorch [35], which relies on a simple fan out algorithm.

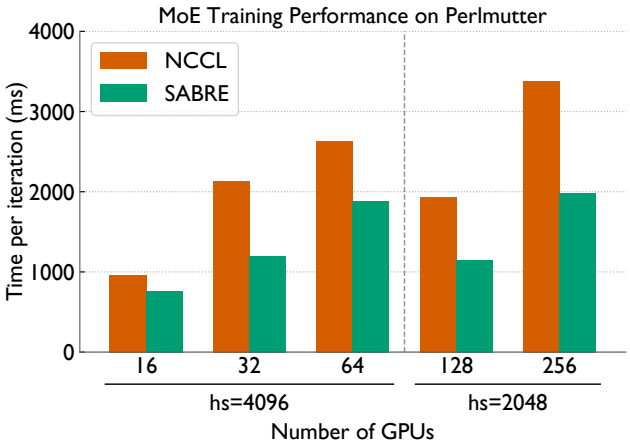


Figure 10: End-to-end MoE training time comparison between SABRE and the default PyTorch implementation.

In our setup, we enable expert parallelism, assign one expert to each GPU, and set the routing to top $k = 2$. Note that MoE routing decisions change every iteration; the communication matrix differs from call to call. SABRE recomputes the MTM ratio and selects the algorithm on every all-to-allv invocation. This dynamic selection overhead is included in the reported end-to-end times. For the 16, 32, and 64 GPU configurations, we keep the hidden size at 4096. Note that the MoE gating mechanism routes most tokens to a small number of experts. As a result, when we scale to 128 and 256 GPUs, the same model size no longer fits in GPU memory, so in these two configurations we halve the model size for each expert. As shown in Figure 10, compared with the NCCL-based PyTorch `all_to_all_single`, SABRE speeds up MoE training by 1.28×, 1.79×, and 1.40× on 16, 32, and 64 GPUs, and by 1.69× and 1.71× on 128 and 256 GPUs. These results demonstrate that our all-to-allv optimizations bring significant end-to-end performance gains for real MoE training workloads.

9 Conclusion

In this paper, we revisit all-to-allv as a key communication bottleneck in deep learning workloads that run on modern GPU systems. We analyze how severe load imbalance and inter-node congestion limit existing MPI and NCCL implementations, and derive a tight lower bound for highly-skewed all-to-allv that is achieved when each node spreads its send and receive data evenly across all NICs. Based on this analysis, we propose a skew-aware 2D all-to-allv algorithm that uses high-bandwidth intra-node links for gathering and redistribution while reducing inter-node contention through grouped, batched communication. For lightly-skewed all-to-allv, we design a 2D all-to-allv pipeline algorithm that overlaps intra-node forwarding with inter-node exchange and actively uses fast intra-node links to relieve network pressure. At runtime, SABRE uses the maximum-to-mean ratio to select the most suitable all-to-allv algorithm. SABRE also provides a Python interface that can serve as a direct replacement for `dist.all_to_all_single` in deep learning frameworks. Experiments on the Perlmutter supercomputer demonstrate that SABRE achieves up to 2.4× speedup over Cray MPICH and NCCL in microbenchmarks, and delivers up to 1.8× speedup in end-to-end Megatron-LM MoE training compared with the default PyTorch implementation.

Acknowledgments

The authors thank Siddharth Singh (NVIDIA, Inc.) for his valuable feedback. This material is based upon work supported by the National Science Foundation under Grant No. 2047120. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy (DOE) Office of Science User Facility, operated under Contract No. DE-AC02-05CH11231 using NERSC awards DDR-ERCAP0034262 and ALCC-ERCAP0034775.

References

[1] 2020. NCCL. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/overview.html

[2] 2024. HPE Cassini Performance Counters. https://cpe.ext.hpe.com/docs/latest/getting_started/HPE-Cassini-Performance-Counters.html

[3] 2025. RCCL. https://github.com/ROCm/rccl

[4] Abhinav Bhatele, Todd Gamblin, Steven H. Langer, Peer-Timo Bremer, Erik W. Draeger, Bernd Hamann, Katherine E. Isaacs, Aaditya G. Landge, Joshua A. Levine, Valerio Pascucci, Martin Schulz, and Charles H. Still. 2012. Mapping Applications with Collectives over Sub-communicators on Torus Networks. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society. http://doi.ieeecomputersociety.org/10.1109/SC.2012.75 LLNL-CONF-556491.

[5] Jehoshua Bruck, Ching-Tien Ho, Shlomo Kipnis, and Derrick Weathersby. 1994. Efficient algorithms for all-to-all communications in multi-port message-passing systems. In *Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*. 298–309.

[6] 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* (Virtual Event, Republic of Korea) *(PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 62–75. doi:10.1145/3437801.3441620

[7] 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.

[8] Chen-Chun Chen, Kawthar Shafie Khorassani, Quentin G Anthony, Aamir Shafi, Hari Subramoni, and Dhabaleswar K Panda. 2022. Highly efficient alltoall and alltoallv communication algorithms for gpu systems. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 24–33.

[9] Chen-Chun Chen, Kawthar Shafie Khorassani, Pouya Kousha, Qinghua Zhou, Jinghan Yao, Hari Subramoni, and Dhabaleswar K Panda. 2023. Mpi-xccl: A portable mpi library over collective communication libraries for various accelerators. In *Proceedings of the SC'23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*. 847–854.

[10] Harunobu Daikoku, Hideyuki Kawashima, and Osamu Tatebe. 2019. Skew-aware collective communication for MapReduce shuffling. *IEICE TRANSACTIONS on Information and Systems* 102, 12 (2019), 2389–2399.

[11] Daniele De Sensi, Salvatore Di Girolamo, Kim H McMahon, Duncan Roweth, and Torsten Hoefler. 2020. An in-depth analysis of the slingshot interconnect. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[12] Daniele De Sensi, Lorenzo Pichetti, Flavio Vella, Tiziano De Matteis, Zebin Ren, Luigi Fusco, Matteo Turisini, Daniele Cesarini, Kurt Lust, Animesh Trivedi, Duncan Roweth, Filippo Spiga, Salvatore Di Girolamo, and Torsten Hoefler. 2024. Exploring GPU-to-GPU Communication: Insights into Supercomputer Interconnects. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Atlanta, GA, USA) *(SC '24)*. IEEE Press, Article 33, 15 pages. doi:10.1109/SC41406.2024.00039

[13] Ke Fan, Jens Domke, Seydou Ba, and Sidharth Kumar. 2025. Parameterized Algorithms for Non-uniform All-to-all. In *Proceedings of the 34th International Symposium on High-Performance Parallel and Distributed Computing*. 1–13.

[14] Ke Fan, Thomas Gilray, and Sidharth Kumar. [n. d.]. Padding to Extend the Bruck Algorithm for Non-uniform All-to-all Communication. ([n. d.]).

[15] Ke Fan, Thomas Gilray, Valerio Pascucci, Xuan Huang, Kristopher Micinski, and Sidharth Kumar. 2022. Optimizing the bruck algorithm for non-uniform all-to-all communication. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*. 172–184.

[16] Ke Fan, Steve Petruzza, Thomas Gilray, and Sidharth Kumar. 2024. Configurable algorithms for all-to-all collectives. In *ISC High Performance 2024 Research Paper Proceedings (39th International Conference)*. Prometeus GmbH, 1–12.

[17] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research* 23, 120 (2022), 1–39.

[18] M Frigo and S G Johnson. 2005. The Design and Implementation of FFTW3. *Proc. IEEE* 93, 2 (2005), 216–231.

[19] Trevor Gale, Deepak Narayanan, Cliff Young, and Matei Zaharia. 2023. Megablocks: Efficient sparse training with mixture-of-experts. *Proceedings of Machine Learning and Systems* 5 (2023), 288–304.

[20] Richard L. Graham, Timothy S. Woodall, and Jeffrey M. Squyres. 2005. Open MPI: A Flexible High Performance MPI. In *Proceedings, 6th Annual International Conference on Parallel Processing and Applied Mathematics*. Poznan, Poland.

[21] William Gropp, Ewing (Rusty) Lusk, Rajeev Thakur, Pavan Balaji, Thomas Gillis, Yanfei Guo, Rob Latham, Ken Raffenetti, and Hui Zhou. 2023. MPICH. [Computer Software] https://doi.org/10.11578/dc.20200514.13. doi:10.11578/dc.20200514.13

[22] 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.

[23] Sascha Hunold, Abhinav Bhatele, George Bosilca, and Peter Knees. 2020. Predicting MPI Collective Communication Performance Using Machine Learning. In *Proceedings of the IEEE Cluster Conference (Cluster '20)*. doi:10.1109/CLUSTER49012.2020.00036

[24] Changho Hwang, Peng Cheng, Roshan Dathathri, Abhinav Jangda, Saeed Maleki, Madan Musuvathi, Olli Saarikivi, Aashaka Shah, Ziyue Yang, Binyang Li, et al. 2026. MSCCL++: Rethinking GPU Communication Abstractions for AI Inference. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 1201–1215.

[25] Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Prabhat Ram, et al. 2023. Tutel: Adaptive mixture-of-experts at scale. *Proceedings of Machine Learning and Systems* 5 (2023), 269–287.

[26] Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. 2024. Mixtral of experts. *arXiv preprint arXiv:2401.04088* (2024).

[27] J. Kim, W. J. Dally, S. Scott, and D. Abts. 2008. Technology-Driven, Highly-Scalable Dragonfly Topology. In *2008 International Symposium on Computer Architecture*. IEEE Computer Society.

[28] Yiran Lei, Dongjoo Lee, Liangyu Zhao, Daniar Kurniawan, Chanmyeong Kim, Heetaek Jeong, Changsu Kim, Hyeonseong Choi, Liangcheng Yu, Arvind Krishnamurthy, et al. 2025. FAST: An Efficient Scheduler for All-to-All GPU Communication. *arXiv preprint arXiv:2505.09764* (2025).

[29] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2020. GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding. doi:10.48550/ARXIV.2006.16668

[30] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[31] 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* (Sydney, NSW, Australia) *(ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 16–37. doi:10.1145/3651890.3672249

[32] NERSC. [n. d.]. Perlmutter System Architecture. https://docs.nersc.gov/systems/perlmutter/architecture/.

[33] Xiaonan Nie, Pinxue Zhao, Xupeng Miao, Tong Zhao, and Bin Cui. 2022. HetuMoE: An Efficient Trillion-scale Mixture-of-Expert Distributed Training System. doi:10.48550/ARXIV.2203.14685

[34] NVIDIA. [n. d.]. NVLink and NVSwitch: The Building Blocks of Advanced Multi-GPU Communication. https://www.nvidia.com/en-us/data-center/nvlink/.

[35] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[36] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. 2022. Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale. In *International conference on machine learning*. PMLR, 18332–18346.

[37] Andres Sewell, Ke Fan, Ahmedur Rahman Shovon, Landon Dyken, Sidharth Kumar, and Steve Petruzza. 2024. Bruck algorithm performance analysis for multi-GPU all-to-all communication. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*. 127–133.

[38] 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.

[39] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. *Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism.* Technical Report. arXiv:1909.08053 [cs.CL]

[40] Siddharth Singh, Olatunji Ruwase, Ammar Ahmad Awan, Samyam Rajbhandari, Yuxiong He, and Abhinav Bhatele. 2023. A Hybrid Tensor-Expert-Data Parallelism Approach to Optimize Mixture-of-Experts Training. In *Proceedings of the 37th International Conference on Supercomputing* (Orlando, FL, USA) *(ICS '23)*. Association for Computing Machinery, New York, NY, USA, 203–214. doi:10.1145/3577193.3593704

[41] 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.

[42] Jesper Larsson Träff, Antoine Rougier, and Sascha Hunold. 2014. Implementing a classic: Zero-copy all-to-all communication with MPI datatypes. In *Proceedings of the 28th ACM international conference on Supercomputing*. 135–144.

[43] Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G Andersen, Gregory R Ganger, Garth A Gibson, and Brian Mueller. 2009. Safe and effective fine-grained TCP retransmissions for datacenter communication. *ACM SIGCOMM computer communication review* 39, 4 (2009), 303–314.

[44] Heng Xu, Zhiwei Yu, Chengze Du, Ying Zhou, Letian Li, Haojie Wang, Weiqiang Cheng, and Jialong Li. 2025. RailS: Load Balancing for All-to-All Communication in Distributed Mixture-of-Experts Training. *arXiv preprint arXiv:2510.19262* (2025).

[45] Shulai Zhang, Ningxin Zheng, Haibin Lin, Ziheng Jiang, Wenlei Bao, Chengquan Jiang, Qi Hou, Weihao Cui, Size Zheng, Li-Wen Chang, et al. 2025. Comet: Fine-grained computation-communication overlapping for mixture-of-experts. *Proceedings of Machine Learning and Systems* 7 (2025).