RAILWAVE: ADAPTIVE SPATIAL AND TEMPORAL SCHEDULING FOR EXPERT-PARALLEL COMMUNICATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Irregular All-to-All communication is a major bottleneck in expert-parallel Mixture-of-Experts (MoE) models. Even with fixed expert routing and placement, uneven utilization of parallel network Rails and incast can limit communication performance. We present RailWave, a phase-adaptive communication layer built on DeepEP that addresses these bottlenecks below the routing layer through spatial and temporal traffic shaping. RailBalance redistributes source traffic across eligible Rails using source-local information, while a reusable, topology-derived permutation schedule limits concurrent senders per receiver without rebuilding demand-dependent schedules for each communication phase. A lightweight calibrated selector chooses an execution path according to each phase's traffic characteristics and offline profiling results. On training-derived communication workloads from the 106B GLM-4.5-Air model, RailWave delivers up to $5.84\times$ speedup on H800 and $4.36\times$ on H20 over Native.

1 INTRODUCTION

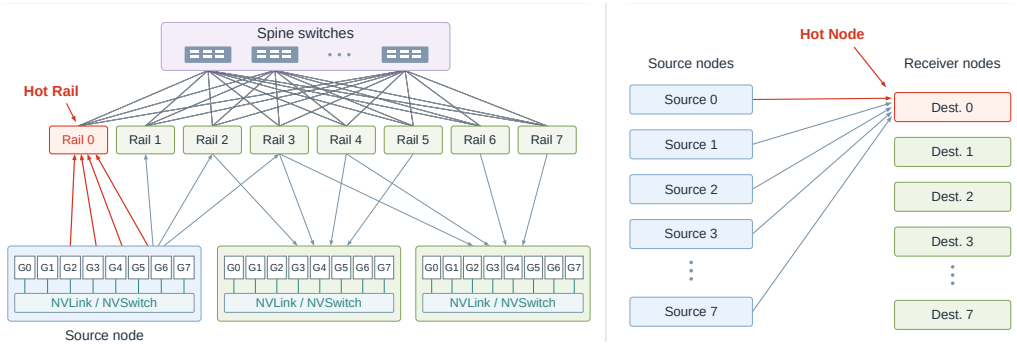
Mixture-of-Experts (MoE) models increase model capacity while activating only a subset of parameters for each token (Shazeer et al., 2017; Lepikhin et al., 2021; Fedus et al., 2022). Expert parallelism (EP) distributes experts across GPUs, requiring dispatch and combine exchanges between tokens and their selected experts. Because routing induces uneven and input-dependent traffic, these exchanges form irregular All-to-All communication and can become a major bottleneck in MoE execution (Liao et al., 2025; Li et al., 2023).

Existing approaches improve MoE efficiency through routing, expert placement, and replication (Lewis et al., 2021; Zhou et al., 2022; Nie et al., 2023; He et al., 2022). Replication-based methods can additionally incur per-layer memory overhead from redundant expert states (DeepSeek-AI, 2025). These mechanisms shape the logical communication demand, but do not fully determine its physical execution. This distinction is particularly important in Rail-optimized clusters, where a high-bandwidth intra-node fabric connects multiple scale-out paths (NVIDIA, 2023; Wang et al., 2024). Even with an optimized dispatch–combine backend such as DeepEP (Zhao et al., 2025), traffic can still be redistributed across eligible Rails and node-pair transfers can overlap in different orders.

These execution choices expose two bottlenecks, as shown in Figure 1: (a) Balanced expert and destination-node loads can coexist with concentrated source-Rail traffic, leaving parallel paths underutilized. (b) Conversely, balanced source Rails do not prevent multiple senders from converging on the same receiver and creating incast. Our analysis of 900 training step–layer pairs further shows that expert-load imbalance has limited linear explanatory power for Rail imbalance. We therefore jointly control where traffic enters the scale-out network and which node-pair transfers overlap.

We present RailWave, a communication layer built on DeepEP for spatial and temporal control of expert-parallel traffic. RailBalance redistributes outgoing traffic across eligible Rails using source-local information, without requiring destination-side balancing. Reusable topology-derived permutation scheduling organizes node-pair transfers into communication waves that bound incast degree without rebuilding a demand-dependent schedule for each phase.

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



(a) Rail imbalance (b) Receiver incast

Figure 1: Two bottlenecks with fixed routing. (a) Balanced expert and destination-node loads can coexist with a hot source Rail. (b) Balanced source Rails can still feed the same receiver concurrently.

The two mechanisms are complementary: traffic that is balanced across Rails over an entire phase can become concentrated within an individual wave. RailWave therefore exposes four execution paths combining these mechanisms and uses a lightweight calibrated selector to choose among them according to the current traffic characteristics and validated offline profiling results.

This work makes four contributions:

- We present RailWave, a DeepEP-based communication layer for joint control of Rail allocation and node-pair overlap.
- We design RailBalance, a source-local mechanism for shaping traffic across parallel Rails.
- We adapt reusable topology-derived permutation scheduling to bound incast degree across changing routing phases.
- We evaluate RailWave on two 32-GPU H800 and H20 clusters using real GLM-4.5-Air training workloads, achieving up to $5.84\times$ and $4.36\times$ communication speedups, respectively, over Native execution.

2 Preliminaries

2.1 Expert-parallel communication

Under EP, *dispatch* sends routed token representations to remote experts, and *combine* returns their outputs. Routing and expert placement determine the participants and volume of these irregular All-to-All exchanges.

We consider an EP group spanning N nodes. In one dispatch or combine operation, $D_{s,d}$ denotes the traffic volume from source node s to destination node d. We collect these traffic volumes in the node-level demand matrix

$$D = [D_{s,d}]_{s,d=1}^{N} \in \mathbb{R}_{\geq 0}^{N \times N}. \tag{1}$$

Routing, expert placement, and token-to-replica assignment fix this demand before execution. RailWave changes neither D nor the token's destination expert. Instead, it controls the local egress path and the concurrency of node-pair transfers using the DeepEP backend (Zhao et al., 2025).

2.2 Rail-optimized networks and traffic descriptors

Each node contains G accelerators connected by a high-bandwidth intra-node fabric and exposes R scale-out Rails. We index the Rails by $r \in \{0, \ldots, R-1\}$ (NVIDIA, 2023; Wang et al., 2024). Local redistribution can move traffic to another eligible egress Rail before inter-node transmission.

For node pair (s, d), let $\chi_{s,d}(r)$ map source egress Rail r to the destination ingress Rail. We call the topology *Rail-symmetric* when this mapping is a fixed bijection for every node pair. With matching Rail indices, the correspondence reduces to

$$\chi_{s,d}(r) = r. \tag{2}$$

Let $C_{s,r,d}$ count tokens assigned to source egress Rail r before redistribution and routed to node d. Counts are aggregated according to the GPU-to-egress-Rail mapping, which may assign multiple GPUs to one Rail. The eligible source Rails for node pair (s, d) satisfy

$$\mathcal{R}_{s,d} \subseteq \{0, \ldots, R-1\}. \tag{3}$$

We characterize each communication phase using three traffic descriptors that capture its scale, source-Rail imbalance, and receiver-side concentration. We define the average remote payload per GPU as

$$B = \frac{\sum_{s \neq d} D_{s,d}}{NG}. \tag{4}$$

Let $L_{s,r}$ denote the outgoing inter-node traffic carried by Rail r of source node s. We define the global source-Rail skew as

$$\rho_g = \frac{\max_{s,r} L_{s,r}}{\frac{1}{NR}\sum_{s,r} L_{s,r}}. \tag{5}$$

A value of one indicates equal load across all source Rails.

Receiver-side concentration depends on which transfers overlap. For a sending window w, let A_w denote its active directed node pairs. The traffic received by node d in that window is

$$R_{w,d} = \sum_{s:(s,d)\in A_w} D_{s,d}. \tag{6}$$

We then define receiver concentration as

$$\kappa = \max_w \frac{\max_d R_{w,d}}{\frac{1}{N}\sum_d R_{w,d}}. \tag{7}$$

Thus, B captures communication scale, ρ_g captures source-Rail imbalance, and κ captures receiver concentration under the actual sending windows.

2.3 Communication bottlenecks with fixed routing

Figure 1 illustrates the distinction: source-side redistribution addresses Rail imbalance, whereas transfer ordering controls incast.

To examine the first distinction, we analyze routing records from 900 training step–layer pairs and derive Rail loads using the expert-to-Rail mapping. Figure 2(a) selects six cases whose expert-load max/mean ratios lie between 8.2 and 8.4. Their busiest destination-Rail assignment shares range from 15.5% to 24.1%. Across all records, linear fits from expert-load imbalance to source- and destination-Rail imbalance yield $R^2 = 0.019$ and 0.296, respectively (Figure 2(b)). Expert-load imbalance therefore has limited linear explanatory power for the observed Rail-load variation.

Figure 2(c) quantifies the impact of incast on communication latency. We keep the rank-to-rank demand, payload, backend, and number of communication rounds fixed, while varying which node pairs transmit concurrently. The sum of per-layer P50 dispatch–combine latencies increases from 1.801 to 3.009 s, a 67.1% increase. This result shows that the same rank-to-rank demand can incur higher communication latency under stronger incast.

2.4 MoE communication optimization

MoE communication has been optimized at several layers of the software stack. Upper-layer methods change the demand presented to communication: EPLB, UltraEP, MoonEP, and TAOT adjust expert placement, replication, or token-to-replica assignment to balance expert or device loads (DeepSeek-AI, 2025; Wei et al., 2026; Chen et al., 2026; Zhang et al., 2026). NetMoE adjusts input-sample placement to exploit network locality (Liu et al., 2025), while node-limited routing restricts

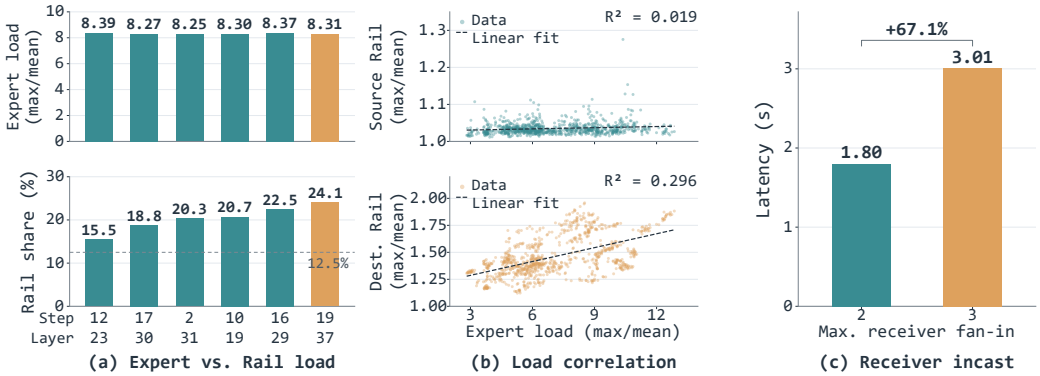


Figure 2: Rail-load imbalance and incast under fixed routing. (a) Six cases with similar expert-load imbalance but different destination-Rail concentration; the two label rows identify training step and layer. (b) Linear fits over 900 training step–layer pairs. (c) Native latency under two replay orders with identical rank-to-rank demand.

the destinations selected for each token (DeepSeek-AI, 2024). These decisions shape the logical demand matrix D. RailWave operates after them and optimizes how the resulting communication is executed.

EP communication libraries optimize the dispatch–combine data path across intra- and inter-node fabrics. DeepEP and NCCL-EP use high-bandwidth intra-node communication to stage or redistribute tokens before scale-out transmission (Zhao et al., 2025; Goldman et al., 2026). SwiftEP reduces data-path overhead through buffer fusion and TMA offloading (Li et al., 2026), while UBEP combines finer-grained execution with topology-aware scheduling for high-bandwidth superpods (Liu et al., 2026). EPIC spans logical and physical optimization, coupling expert migration with topology-adaptive transport and communication–computation overlap (Cao et al., 2026). RailWave builds on DeepEP to control the Rail distribution and concurrent execution of the transfers carried by this backend.

Network-aware mechanisms are the closest to RailWave. RailS and FuseLink primarily improve path utilization through local traffic redistribution (Xu et al., 2026; Ren et al., 2025), whereas RailWave additionally controls which node-pair transfers overlap at a receiver. FAST explicitly balances traffic across the GPUs and NICs within each node, then uses demand-dependent Birkhoff scheduling to construct balanced one-to-one stages (Lei et al., 2026; Wu et al., 2026). This planning requires a consistent view of source–destination demand across nodes; synchronizing that information and rebuilding schedules can incur substantial overhead as routing changes. RailWave instead reuses topology-derived cyclic peer plans with source-local RailBalance, reducing this coordination and plan-construction overhead. Rate-based scheduling controls incast through sender pacing (Cohen et al., 2026). SABRE selects communication strategies according to endpoint-volume skew (Wei & Bhatele, 2026), whereas RailWave indexes offline-calibrated path decisions by payload, global source-Rail skew, and receiver concentration.

Other approaches reduce payload size or overlap communication with computation (Nie et al., 2024; Jiang et al., 2024; Zhang et al., 2025). These optimizations are complementary to RailWave's execution control.

3 System Design

RailWave combines source-local Rail redistribution with node-pair scheduling on top of DeepEP. Figure 3 shows its three-stage workflow: preparation, selection, and execution. Offline profiling produces a cost table, while initialization constructs reusable peer plans from the EP topology. During execution, routing and window-derived traffic statistics feed a selector that chooses among Native, Rail-only, Permutation-only, and Joint. Native retains the backend execution path; Rail-only changes the egress allocation; Permutation-only changes which node pairs communicate

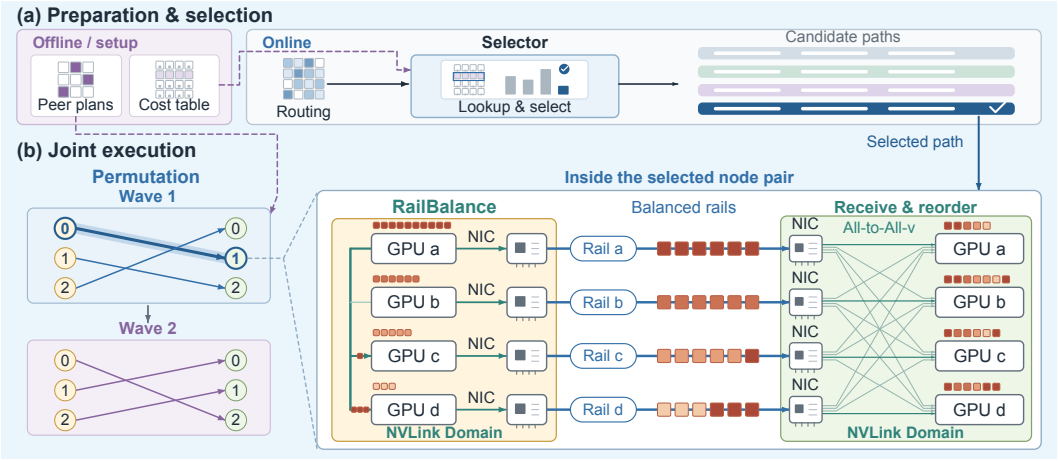


Figure 3: RailWave overview. Offline profiling and current-phase traffic statistics drive execution selection, while topology-derived peer plans are reused across phases. The highlighted Joint path applies RailBalance within scheduled node pairs.

concurrently; Joint applies RailBalance within each scheduled wave. The selector reuses validated offline decisions near calibrated operating points and otherwise falls back to Joint.

The selected plan records the transfer order and owner-to-proxy mapping. Dispatch delivers the original expert inputs through this plan, and combine uses the saved metadata to return outputs to their original token positions. All ranks follow the same active execution path and wave order, and the selected plan remains fixed for the corresponding dispatch–combine pair.

3.1 Source-Local RailBalance

RailBalance redistributes the traffic of each node pair (s, d) across its eligible Rails $\mathcal{R}_{s,d}$. It uses the source counts $C_{s,r,d}$ and fixed egress–ingress correspondence $\chi_{s,d}$ from Section 2.2. If source Rail r carries $x_{s,r,d}$ transfer records, its contribution to the corresponding destination Rail is

$$I_{s,d,\chi_{s,d}(r)} = x_{s,r,d}. \tag{8}$$

The source's ingress contribution is therefore a permutation of its egress allocation. Source-local balancing shapes both ends of a transfer without a destination-side quota exchange. This guarantee concerns each source's contribution; aggregate equality also depends on the eligible sets and the distribution of integer remainders across sources.

Let $B_{s,d} = \sum_r C_{s,r,d}$ be the record count and $m_{s,d} = |\mathcal{R}_{s,d}|$. RailBalance assigns quotas satisfying

$$Q_{s,r,d} \in \left\{ \left\lfloor \frac{B_{s,d}}{m_{s,d}} \right\rfloor, \left\lceil \frac{B_{s,d}}{m_{s,d}} \right\rceil \right\}, \qquad \sum_{r \in \mathcal{R}_{s,d}} Q_{s,r,d} = B_{s,d}. \tag{9}$$

Quotas are zero outside the eligible set. Remainder ownership rotates deterministically across destination slices, and eligible quotas differ by at most one record. Counts retain the backend's per-destination transfer accounting, including any existing token coalescing.

Records within quota remain on their owner Rail. Algorithm 1 deterministically matches surplus records to deficit slots using the shared count and quota vectors, yielding the owner-to-proxy mapping without atomic slot allocation. Only nonidentity mappings require relocation; token positions and expert metadata are retained for combine.

Algorithm 1 Source-local RailBalance plan construction

Require: Source counts $C_{s,r,d}$, eligible Rails $\mathcal{R}_{s,d}$
Ensure: Owner-to-proxy mapping $M_{s,d}$
$B_{s,d} \leftarrow \sum_r C_{s,r,d}$
$m_{s,d} \leftarrow |\mathcal{R}_{s,d}|$
$Q_{s,:,d} \leftarrow \textsc{BalancedQuotas}(B_{s,d}, \mathcal{R}_{s,d}, d)$ ▷ Deterministic remainder rotation
$\mathcal{S} \leftarrow \emptyset$, $\mathcal{D} \leftarrow \emptyset$
for each $r \in \mathcal{R}_{s,d}$ **do**
    Keep the first $\min(C_{s,r,d}, Q_{s,r,d})$ records on r
    if $C_{s,r,d} > Q_{s,r,d}$ **then**
        Append excess records on r to $\mathcal{S}$
    else if $C_{s,r,d} < Q_{s,r,d}$ **then**
        Append deficit slots on r to $\mathcal{D}$
    end if
end for
Sort $\mathcal{S}$ by (owner Rail, local index)
Sort $\mathcal{D}$ by (target Rail, slot index)
for $i = 1, \ldots, |\mathcal{S}|$ **do**
    Map surplus record $\mathcal{S}[i]$ to the target Rail of $\mathcal{D}[i]$
end for
return $M_{s,d}$

3.2 Reusable Permutation Scheduling

Permutation scheduling bounds incast degree using cyclic pairwise exchange (Thakur et al., 2005). With N nodes, wave $k \in \{1, \ldots, N-1\}$ uses

$$\begin{aligned} \pi_k(s) &= (s + k) \bmod N, \\ \pi_k^{-1}(d) &= (d - k) \bmod N. \end{aligned} \tag{10}$$

Each node has at most one outgoing and one incoming peer in a wave. Every remote ordered pair appears exactly once because its shift is uniquely determined by $(d - s) \bmod N$. These peer relationships depend only on group membership and are constructed once; each phase supplies new transfer records and counts.

For a wave, the executor selects records whose destination is $\pi_k(s)$ and packs their payload without padding the node-pair demand. This partition operates over backend transfer records, including multiple destinations for the same token. Within each backend invocation, DeepEP and Rail-only admit the submitted remote pairs together, whereas Permutation-only and Joint execute the cached peer plans in order. Within each wave, disjoint node pairs communicate concurrently.

Joint applies RailBalance to the records in the current wave. A phase can be balanced across Rails in aggregate while a particular destination slice remains concentrated on a few Rails. Redistributing that slice can shorten the wave's critical transfer. The gain depends on its Rail skew and relocation cost; scheduling still serializes the waves and retains the original differences in node-pair demand.

3.3 Calibrated Execution Selection

No single execution path dominates across communication phases. RailBalance is beneficial when improved Rail utilization outweighs local redistribution, while permutation scheduling is beneficial when reduced incast outweighs the cost of additional communication rounds. These break-even points vary with traffic characteristics, motivating phase-specific execution selection.

The selector uses three traffic descriptors defined in Section 2.2: the average remote payload per GPU B, the global source-Rail skew ρ_g, and the receiver concentration κ under the actual sending windows. Together, they capture communication scale, source-Rail imbalance, and receiver concentration for the current phase. After fixed normalization calibrated offline, these statistics form the lookup feature $x = (\widetilde{B}, \widetilde{\rho}_g, \widetilde{\kappa})$. Phases with no remote payload retain the Native local path.

Offline profiling constructs a calibrated anchor table. Each anchor stores a feature vector x_j and an execution path p_j. An alternative to Joint is retained only when repeated measurements are

sufficiently stable and show a conservative advantage over Joint. This filters out decisions whose apparent benefit may be explained by measurement variation. The exact normalization, anchor-retention criteria, and resulting anchors are reported in Appendix A.3.

At runtime, path decisions are reused only within a calibrated neighborhood of these anchors. Let ϵ_m denote the neighborhood radius of feature dimension m. The eligible anchor set is

$$\mathcal{A}(x) = \{j : |x_m - x_{j,m}| \le \epsilon_m,\ \forall m\}. \tag{11}$$

The runtime path decision is

$$p(x) = \begin{cases} p_{j^*}, & j^* = \arg\min_{j \in \mathcal{A}(x)} \|x - x_j\|_2^2, \\ \text{Joint}, & \mathcal{A}(x) = \varnothing. \end{cases} \tag{12}$$

Thus, requests close to a validated operating point reuse its measured path decision, while requests outside all calibrated neighborhoods conservatively fall back to Joint. This guarded lookup avoids extrapolating path choices into traffic regions not supported by offline measurements.

4 Experiments

We evaluate RailWave against Native, NCCL, and the released FAST implementation (Lei et al., 2026), isolate the effects of spatial and temporal shaping, and evaluate adaptive execution. Additional results on integration with an upper-layer expert-parallel system are provided in Appendix A.2.

4.1 Experimental Setup

Platforms. We evaluate on two 32-GPU clusters equipped with NVIDIA H800 and H20 GPUs, respectively. Each cluster has four nodes with eight GPUs per node. All compared methods use the same communication configuration within each cluster. Detailed hardware and software settings are provided in Appendix A.1.

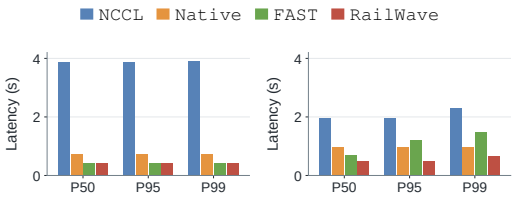
Workloads. Our replay is based on expert-routing statistics collected during actual GLM-4.5-Air (GLM-4.5 Team, 2025) training runs on DAPO-Math-17k (Yu et al., 2025), covering all 45 MoE layers. To model incast under variable rank progress, we construct three controlled sending patterns—Ring, Moderate, and Extreme—as detailed in Appendix A.6.

Baselines and metrics. We compare four execution paths on the same DeepEP backend: Native, Rail-only, Permutation-only, and Joint. Native and Rail-only retain the replayed transfer order, whereas Permutation-only and Joint use cyclic waves; Joint additionally applies RailBalance within each wave. RailWave selects among these paths using the calibrated lookup described in Section 3.3. We also compare against NCCL `all_to_all_single` and the released FAST implementation. The plotted latency is the GPU-event communication interval: all waves of dispatch–combine for the DeepEP paths, or the corresponding forward–return All-to-All for NCCL and FAST, including RailBalance GPU operations when enabled. CPU planning, index construction, and prepacking are outside this interval for every method, as detailed in Appendix A.6. Each plotted P50/P95/P99 is the sum of the corresponding per-layer quantiles over the 45-layer replay.

4.2 Overall Performance

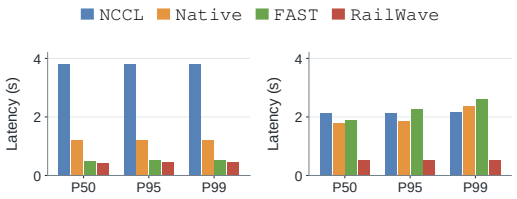
Communication performance. Figures 4–6 compare RailWave with Native, NCCL, and FAST under Ring, Moderate, and Extreme, respectively. RailWave achieves 2.02–$5.84\times$ speedups over Native at P50 on H800 and 1.74–$4.36\times$ on H20 across the three replay patterns, with gains persisting at P95 and P99.

Native slows from Moderate to Extreme under stronger incast, whereas RailWave maintains nearly identical P50 latency across the two patterns.



(a) H20 (b) H800

Figure 4: Communication performance under Ring on H20 and H800.



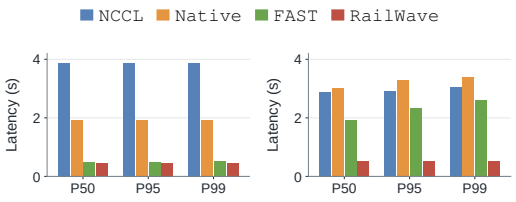
(a) H20 (b) H800

Figure 5: Communication performance under Moderate on H20 and H800.

4.3 Ablation and Mechanism Analysis

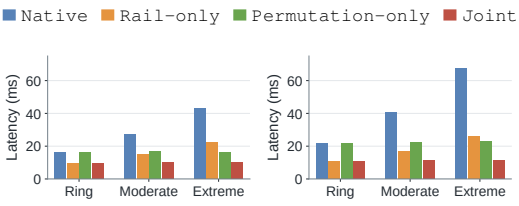
Spatial and temporal contributions. Figure 7 separates the contributions of RailBalance and permutation scheduling. In Ring, Rail-only closely matches Joint while Permutation-only remains close to Native, showing that spatial shaping provides most of the gain when incast is already limited.

Under Moderate and Extreme, combining the two mechanisms improves performance on both clusters. On H800, Joint achieves $1.44\times$ and $1.97\times$ speedups over the better single mechanism, respectively. RailBalance improves Rail utilization, whereas permutation scheduling controls incast.



(a) H20 (b) H800

Figure 6: Communication performance under Extreme on H20 and H800.



(a) H20 (b) H800

Figure 7: Ablation of RailBalance and permutation scheduling on H20 and H800.

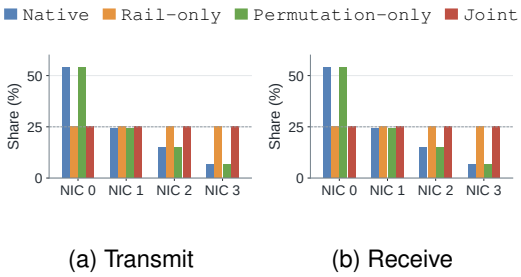
Physical NIC load distribution. Figure 8 shows the effect of RailBalance under Moderate on H20. Native distributes transmitted traffic as approximately 54.12%, 24.33%, 14.85%, and 6.69% across the four NIC positions, whereas Rail-only and Joint produce approximately equal shares in both directions. Permutation-only retains the uneven distribution.

These results show that RailBalance changes physical Rail utilization, whereas permutation scheduling changes which node-pair transfers overlap.

Performance across replay-derived traffic conditions. We further select cases from the replay that span different levels of Rail skew and receiver concentration. Figure 9 shows that when both are high, Joint outperforms either single mechanism. In one such case, Joint reduces mean latency to 4.11 ms, compared with 10.96 ms for Permutation-only and 73.28 ms for Rail-only. More balanced cases can instead favor a lighter execution path, motivating adaptive selection.

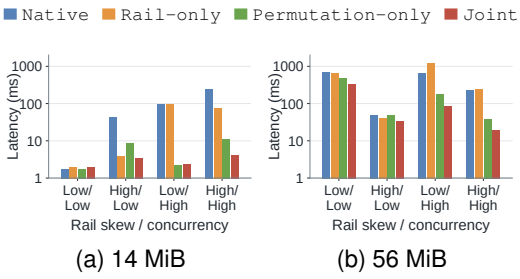
Adaptive execution. We evaluate the calibrated selector on 12 unseen requests that are not used to construct the frozen cost table. Figure 10 shows the four requests where the selector switches away from Joint: three select Permutation-only and one selects Native. All four switches reduce measured median communication latency, by 5.7–16.4%, corresponding to 1.061–$1.196\times$ speedup over Joint. The remaining eight requests fall back to Joint. Complete calibration and validation results, including the development-regression set, are provided in Appendix A.3.

Additional UltraEP integration results, packing measurements, and workspace accounting are provided in Appendices A.2, A.4, and A.5, respectively.



(a) Transmit (b) Receive

Figure 8: Physical NIC traffic distribution under Moderate on H20.



(a) 14 MiB (b) 56 MiB

Figure 9: Execution-path performance across replay-derived cases with different Rail-skew and receiver-concentration characteristics.

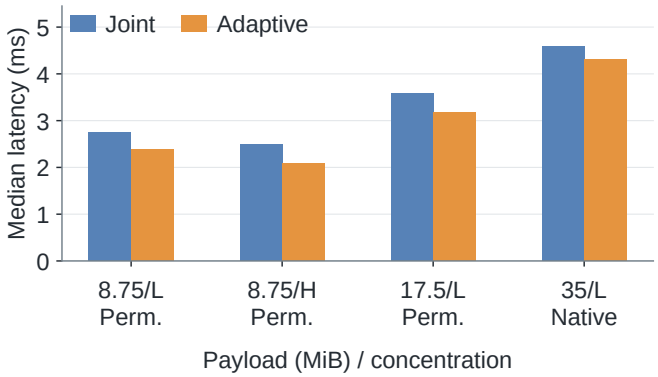


Figure 10: Adaptive execution on unseen requests where the calibrated selector switches away from Joint. L/H denote low/high receiver concentration; the selected execution path is shown below each case.

5 Conclusion

We presented RailWave, a communication layer for expert-parallel MoE execution that targets physical-network inefficiencies remaining after expert routing and placement are fixed. RailWave combines source-local RailBalance with reusable permutation scheduling to jointly improve parallel Rail utilization and control receiver-side concurrency, while preserving expert assignments and logical communication demand. Built on DeepEP, RailWave achieves communication speedups of up to $5.84\times$ on H800 and $4.36\times$ on H20, demonstrating the benefits of jointly shaping the spatial and temporal execution of expert-parallel traffic.

AI use statement

In this work, we used generative AI tools to assist with language polishing, code debugging, and figure revision. All AI-assisted text was reviewed and edited by the authors, code changes were inspected and tested, and figure revisions were verified against the underlying data. We take responsibility for the final content of this work, including all text, code, figures, claims, and other artifacts produced with the aid of generative AI.

Reproducibility statement

We provide the design and implementation details of RailWave in Section 3, and describe the evaluation setup and workloads in Section 4.1. Further details cover hardware and software settings (Appendix A.1), UltraEP integration (Appendix A.2), selector calibration and validation (Appendix A.3), packing overhead (Appendix A.4), memory overhead (Appendix A.5), and the replay protocol (Appendix A.6). An anonymized source-code repository will be provided with the submission to support reproduction of the reported results. The repository will be made public upon publication.

REFERENCES

Jiamin Cao, Qingxu Li, Yaozhong Liu, Jiaqi Gao, Yan Zhang, Shangfeng Shi, Zian Chen, Yizhi Wang, Jun Zhang, Kunling He, Ennan Zhai, Jianbo Dong, Binzhang Fu, and Dennis Cai. Balancing and beyond: Communication-centric optimizations in expert parallelism. In *Proceedings of the ACM SIGCOMM 2026 Conference*, pp. 550–563, 2026. doi: 10.1145/3789240.3829201. URL https://doi.org/10.1145/3789240.3829201.

Yutian Chen, Cong Li, Yucheng Wang, and Ming Wei. MoonEP: A perfectly balanced expert parallelism library via dynamic redundant experts. https://github.com/MoonshotAI/MoonEP, 2026.

Evyatar Cohen, Jose Yallouz, Alexander Shpiner, Mark Silberstein, Sylvia Ratnasamy, and Isaac Keslassy. Incast-free MoE rate-based scheduling. *arXiv preprint arXiv:2607.26340*, 2026. URL https://arxiv.org/abs/2607.26340.

DeepSeek-AI. DeepSeek-V3 technical report. *arXiv preprint arXiv:2412.19437*, 2024. URL https://arxiv.org/abs/2412.19437.

DeepSeek-AI. EPLB: Expert parallelism load balancer. https://github.com/deepseek-ai/EPLB, 2025.

William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022. URL https://www.jmlr.org/papers/v23/21-0998.html.

GLM-4.5 Team. GLM-4.5: Agentic, reasoning, and coding (ARC) foundation models. *arXiv preprint arXiv:2508.06471*, 2025. URL https://arxiv.org/abs/2508.06471.

Amos Goldman, Nimrod Boker, Maayan Sheraizin, Nimrod Admoni, Artem Polyakov, Subhadeep Bhattacharya, Fan Yu, Kai Sun, Georgios Theodorakis, Hsin-Chun Yin, Peter-Jan Gootzen, Aamir Shafi, Assaf Ravid, Salvatore Di Girolamo, James Dinan, Xiaofan Li, Manjunath Gorentla Venkata, and Gil Bloch. NCCL EP: Towards a unified expert parallel communication API for NCCL. *arXiv preprint arXiv:2603.13606*, 2026. URL https://arxiv.org/abs/2603.13606.

Jiaao He, Jidong Zhai, Tiago Antunes, Haojie Wang, Fuwen Luo, Shangfeng Shi, and Qin Li. FasterMoE: Modeling and optimizing training of large-scale dynamic pre-trained models. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 120–134, 2022. doi: 10.1145/3503221.3508418. URL https://doi.org/10.1145/3503221.3508418.

Chenyu Jiang, Ye Tian, Zhen Jia, Shuai Zheng, Chuan Wu, and Yida Wang. Lancet: Accelerating mixture-of-experts training via whole graph computation-communication overlapping. *Proceedings of Machine Learning and Systems*, 6:74–86, 2024. URL https://proceedings.mlsys.org/paper_files/paper/2024/hash/339caf45a6fa281cae8adc6465343464-Abstract-Conference.html.

Yiran Lei, Dongjoo Lee, Liangyu Zhao, Daniar Kurniawan, Chanmyeong Kim, Heetaek Jeong, Changsu Kim, Hyeonseong Choi, Liangcheng Yu, Arvind Krishnamurthy, Justine Sherry, and Eriko Nurvitadhi. FAST: An efficient scheduler for All-to-All GPU communication. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, pp. 2515–2531, Renton, WA, May 2026. USENIX Association. ISBN 978-1-939133-54-0. URL https://www.usenix.org/conference/nsdi26/presentation/lei-yiran.

Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. GShard: Scaling giant models with conditional computation and automatic sharding. In *International Conference on Learning Representations*, 2021. URL https://arxiv.org/abs/2006.16668.

Mike Lewis, Shruti Bhosale, Tim Dettmers, Naman Goyal, and Luke Zettlemoyer. BASE layers: Simplifying training of large, sparse models. In *Proceedings of the 38th International Conference*

on Machine Learning, volume 139 of *Proceedings of Machine Learning Research*, pp. 6265–6274. PMLR, 2021. URL https://proceedings.mlr.press/v139/lewis21a.html.

Jiamin Li, Yimin Jiang, Yibo Zhu, Cong Wang, and Hong Xu. Accelerating distributed MoE training and inference with Lina. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pp. 945–959, Boston, MA, July 2023. USENIX Association. ISBN 978-1-939133-35-9. URL https://www.usenix.org/conference/atc23/presentation/li-jiamin.

Xingyi Li, Yadong Liu, Xiaojie Huang, Yiran Zhang, Shuai Wang, Shangguang Wang, Zhehao Lin, Yinben Xia, Chang Yu, Qihang Liu, Xuan Zhang, Hao Lu, Xiang Li, Zekun He, Yachen Wang, and Xianneng Zou. SwiftEP: Accelerating MoE inference with buffer fusion and TMA offloading. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, pp. 1073–1089, Renton, WA, May 2026. USENIX Association. ISBN 978-1-939133-54-0. URL https://www.usenix.org/conference/nsdi26/presentation/li-xingyi.

Xudong Liao, Yijun Sun, Han Tian, Xinchen Wan, Yilun Jin, Zilong Wang, Zhenghang Ren, Xinyang Huang, Wenxue Li, Kin Fai Tse, Zhizhen Zhong, Guyue Liu, Ying Zhang, Xiaofeng Ye, Yiming Zhang, and Kai Chen. MixNet: A runtime reconfigurable optical-electrical fabric for distributed mixture-of-experts training. In *Proceedings of the ACM SIGCOMM 2025 Conference*, pp. 554–574, 2025. doi: 10.1145/3718958.3750465. URL https://doi.org/10.1145/3718958.3750465.

Xinyi Liu, Yujie Wang, Fangcheng Fu, Xupeng Miao, Shenhan Zhu, Xiaonan Nie, and Bin Cui. NetMoE: Accelerating MoE training through dynamic sample placement. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/e0c256700465c158de71081b4cf5e8c3-Abstract-Conference.html.

Yipeng Liu, Chang Liu, Si Shen, Jiaqi Zheng, Mingfan Li, Yuyang Yang, Guanhua Li, Yuquan Zhang, Yimeng Xu, Zhongzhe Hu, Zhiyuan Huang, Qihang Duan, Junsong Wang, Wenkai Ling, Baochuan Yang, Xianzhi Yu, Han Bao, Yijie Chen, and Guihai Chen. UBEP: Re-architecting expert parallelism communication library for production superpods. In *Proceedings of the ACM SIGCOMM 2026 Conference*, pp. 415–432, 2026. doi: 10.1145/3789240.3829183. URL https://doi.org/10.1145/3789240.3829183.

Xiaonan Nie, Xupeng Miao, Zilong Wang, Zichao Yang, Jilong Xue, Lingxiao Ma, Gang Cao, and Bin Cui. FlexMoE: Scaling large-scale sparse pre-trained model training via dynamic device placement. *Proceedings of the ACM on Management of Data*, 1(1):1–19, 2023. doi: 10.1145/3588964. URL https://doi.org/10.1145/3588964.

Xiaonan Nie, Qibin Liu, Fangcheng Fu, Shenhan Zhu, Xupeng Miao, Xiaoyang Li, Yang Zhang, Shouda Liu, and Bin Cui. LSH-MoE: Communication-efficient MoE training via locality-sensitive hashing. *Advances in Neural Information Processing Systems*, 37:54161–54182, 2024. doi: 10.52202/079017-1716. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/61674667d642ae52f6bb281bea90ee29-Abstract-Conference.html.

NVIDIA. NVIDIA DGX SuperPOD: Next generation scalable infrastructure for AI leadership: Reference architecture featuring NVIDIA DGX H100 systems. Technical Report RA-11333-001 V11, NVIDIA, September 2023. URL https://docs.nvidia.com/dgx-superpod/reference-architecture-scalable-infrastructure-h100/latest/.

Zhenghang Ren, Yuxuan Li, Zilong Wang, Xinyang Huang, Wenxue Li, Kaiqiang Xu, Xudong Liao, Yijun Sun, Bowen Liu, Han Tian, Junxue Zhang, Mingfei Wang, Zhizhen Zhong, Guyue Liu, Ying Zhang, and Kai Chen. Enabling efficient GPU communication over multiple NICs with FuseLink. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pp. 91–108, Boston, MA, July 2025. USENIX Association. ISBN 978-1-939133-47-2. URL https://www.usenix.org/conference/osdi25/presentation/ren.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In

International Conference on Learning Representations, 2017. URL https://openreview.net/forum?id=B1ckMDqlg.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2020. URL https://arxiv.org/abs/1909.08053v4. Version 4.

Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005. doi: 10.1177/1094342005051521. URL https://doi.org/10.1177/1094342005051521.

Weiyang Wang, Manya Ghobadi, Kayvon Shakeri, Ying Zhang, and Naader Hasani. Rail-only: A low-cost high-performance network for training LLMs with trillion parameters. In *2024 IEEE Symposium on High-Performance Interconnects (HOTI)*, pp. 1–10. IEEE, 2024. doi: 10.1109/HOTI63208.2024.00013. URL https://doi.org/10.1109/HOTI63208.2024.00013.

Cunyang Wei and Abhinav Bhatele. Skew-aware adaptive all-to-allv algorithms for dynamic deep learning workloads. In *Proceedings of the 40th ACM International Conference on Supercomputing*, pp. 450–462, 2026. doi: 10.1145/3797905.3800541. URL https://doi.org/10.1145/3797905.3800541.

Xinming Wei, Chao Jin, Tuo Dai, Yinmin Zhong, Shan Yu, Chengxu Yang, Bingyang Wu, Zili Zhang, Jing Mai, Qianchao Zhu, Zhouyang Li, Yuliang Liu, and Guojie Luo. UltraEP: Unleash MoE training and inference on rack-scale nodes with near-optimal load balancing. *arXiv preprint arXiv:2606.04101*, 2026. URL https://arxiv.org/abs/2606.04101.

Yen-Chieh Wu, Cheng-Shang Chang, Duan-Shin Lee, and H. Jonathan Chao. Dynamic hierarchical Birkhoff–von Neumann decomposition for All-to-All GPU communication. *arXiv preprint arXiv:2602.22756*, 2026. URL https://arxiv.org/abs/2602.22756.

Anxhelo Xhebraj, Sean Lee, Hanfeng Chen, and Vinod Grover. Scaling deep learning training with MPMD pipeline parallelism. *arXiv preprint arXiv:2412.14374*, 2024. URL https://arxiv.org/abs/2412.14374v1. Version 1.

Heng Xu, Zhiwei Yu, Chengze Du, Ying Zhou, Letian Li, Haojie Wang, Weiqiang Cheng, and Jialong Li. RailS: Load balancing for all-to-all communication in distributed mixture-of-experts training. *IEEE Transactions on Networking*, 34:4431–4448, 2026. doi: 10.1109/TON.2026.3676382. URL https://doi.org/10.1109/TON.2026.3676382.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Mu Qiao, Yonghui Wu, and Mingxuan Wang. DAPO: An open-source LLM reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025. URL https://arxiv.org/abs/2503.14476.

Lingyun Zhang, Henghua Zhang, Shilei Gu, Kai Mo, Shuai Han, Shiyong Li, Yanpeng Wang, and Dou Shen. TAOT: Topology-aware optimal transport for dynamic expert replica placement in MoE training. *arXiv preprint arXiv:2608.03676*, 2026. URL https://arxiv.org/abs/2608.03676.

Shulai Zhang, Ningxin Zheng, Haibin Lin, Ziheng Jiang, Wenlei Bao, Chengquan Jiang, Qi Hou, Weihao Cui, Size Zheng, Li-Wen Chang, Quan Chen, and Xin Liu. COMET: Fine-grained computation-communication overlapping for mixture-of-experts. *Proceedings of Machine Learning and Systems*, 7, 2025. URL https://proceedings.mlsys.org/paper_files/paper/2025/hash/e27ea0cd50b798ff8942caf9203f0992-Abstract-Conference.html.

Chenggang Zhao, Shangyan Zhou, Liyue Zhang, Chengqi Deng, Zhean Xu, Yuxuan Liu, Kuai Yu, Jiashi Li, and Liang Zhao. DeepEP: An efficient expert-parallel communication library. https://github.com/deepseek-ai/DeepEP, 2025. Official upstream package, version 2.1.0.

Yanqi Zhou, Tao Lei, Hanxiao Liu, Nan Du, Yanping Huang, Vincent Zhao, Andrew M. Dai, Zhifeng Chen, Quoc V. Le, and James Laudon. Mixture-of-experts with expert choice routing. In *Advances in Neural Information Processing Systems*, volume 35, pp. 7103–7114, 2022. URL https://arxiv.org/abs/2202.09368.

A IMPLEMENTATIONS

A.1 HARDWARE AND SOFTWARE CONFIGURATION

Table 1 summarizes the hardware and software configurations of the two clusters used in our evaluation. Both clusters contain four nodes with eight GPUs per node. Unless otherwise specified, the software versions correspond to the real-routing replay experiments.

Table 1: Hardware and software configurations of the evaluation clusters.

Configuration	H800 Cluster	H20 Cluster
Number of nodes	4	4
GPUs per node	8	8
GPU model	NVIDIA H800	NVIDIA H20
GPU memory	81,559 MiB	97,871 MiB
Intra-node interconnect	NVLink	NVLink
Number of NICs per node	8 RDMA HCA	4 RDMA HCA
NIC bandwidth	200 Gb/s per port	200 Gb/s per port
Inter-node network	RDMA; physical fabric type not recorded	RDMA; physical fabric type not recorded
NVIDIA driver	550.90.07	590.48.01
CUDA	12.9	12.8
NCCL	2.30.7	2.31.2
DeepEP	Official upstream base, package v2.1.0	Official upstream base, package v2.1.0
GPU–NIC mapping	One PIX-proximate HCA per GPU	NCCL-managed
RDMA backend	NCCL GIN, RAIL connection	NCCL GIN, RAIL connection
DeepEP communication contexts	129 GIN contexts	129 GIN contexts
Communication SMs	16	16

For the H800 cluster, the archived topology reports one PIX-proximate HCA for each of the eight GPUs. For the H20 cluster, four HCA candidates (`mlx5_0`, `mlx5_3`, `mlx5_4`, and `mlx5_7`) were configured, while the exact runtime GPU–NIC mapping was not recorded. Therefore, we do not infer the number of actively used physical Rails from the configured HCA set.

Native, Rail-only, Permutation-only, and Joint use the same backend, based on the official DeepEP package version 2.1.0, and the same launch-level communication configuration within each paired replay experiment. The DeepEP configuration allocates 129 GIN communication contexts and uses 16 communication SMs with automatic QP selection. External baselines such as NCCL and FAST retain their own communication backends and library-specific settings.

A.2 INTEGRATION WITH ULTRAEP

We further evaluate RailWave when integrated with UltraEP. UltraEP determines expert placement and maps logical expert IDs to physical experts, while RailWave operates below the routing layer and optimizes how the resulting token traffic is executed on the network. RailWave does not modify the physical expert selected by UltraEP.

For each MoE layer, we first invoke the UltraEP manager to perform expert placement, weight synchronization, and logical-to-physical expert rerouting. The resulting physical expert IDs and routing weights are then passed to the DeepEP communication backend. All compared communication modes use the same UltraEP routing results.

Workloads. We construct two synthetic traffic families, both with three source nodes sending to the same destination node. The first introduces *source-node skew*: traffic volumes differ across the three source nodes, while traffic within each source node is approximately balanced across its local Rails. This workload primarily exposes receiver-side contention.

The second introduces *source-Rail skew*: the three source nodes contribute approximately equal traffic volumes, while traffic within each source node is concentrated on a subset of Rails. This workload therefore combines receiver-side contention with source-Rail imbalance.

We evaluate two traffic volumes, corresponding to 65,536 and 98,304 remote tokens. With BF16 activations and a hidden dimension of 7,168, they correspond to average inter-node payloads of

Table 2: UltraEP integration results under source-node skew (N) and source-Rail skew (R). 28 and 42 denote the average inter-node payload in MiB per GPU per All-to-All operation. Speedup is relative to Native using p50 latency. Best values in each case are bold.

Case	Mode	p50 (ms)	p99 (ms)	Speedup
N28	Native	108.478	154.055	1.000×
N28	RailOnly	79.735	153.605	1.360×
N28	Permutation	**11.191**	12.440	**9.694×**
N28	Joint	11.489	**12.138**	9.442×
N42	Native	141.361	192.259	1.000×
N42	RailOnly	76.374	93.600	1.851×
N42	Permutation	**16.305**	19.180	**8.670×**
N42	Joint	16.712	**17.942**	8.459×
R28	Native	138.624	243.630	1.000×
R28	RailOnly	37.467	106.081	3.700×
R28	Permutation	56.899	59.076	2.436×
R28	Joint	**17.632**	**19.506**	**7.862×**
R42	Native	233.130	352.334	1.000×
R42	RailOnly	102.411	130.325	2.276×
R42	Permutation	84.742	84.892	2.751×
R42	Joint	**25.721**	**25.849**	**9.064×**

approximately 28 MiB and 42 MiB per GPU per All-to-All operation, respectively. Logical routing is approximately uniform across the eight experts on the destination node.

UltraEP placement and rerouting are performed before communication. In these workloads, the hotspot experts are already distributed within the destination node, so UltraEP does not move the resulting traffic away from that node.

Timing boundary. The reported communication latency includes DeepEP dispatch and combine, as well as the RailBalance GPU operations when enabled. UltraEP placement, weight synchronization, and rerouting are measured separately and are not included in the communication latency. Destination grouping, prepacking, and output-order reconstruction are also performed outside the communication timing interval.

Communication results. Table 2 reports the communication latency under the two traffic families.

For source-node-skew traffic, temporal scheduling provides most of the benefit. At 28 MiB per GPU, Permutation reduces p50 latency from 108.478 ms to 11.191 ms, corresponding to a $9.694\times$ speedup over Native. At 42 MiB, it reduces latency from 141.361 ms to 16.305 ms, corresponding to an $8.670\times$ speedup. Since the source-side Rails are already approximately balanced in this workload, applying RailBalance on top of permutation scheduling provides little additional benefit.

For source-Rail-skew traffic, spatial and temporal traffic shaping become complementary. At 28 MiB, Joint reduces p50 latency from 138.624 ms to 17.632 ms, corresponding to a $7.862\times$ speedup over Native. At 42 MiB, Joint reduces latency from 233.130 ms to 25.721 ms, corresponding to a $9.064\times$ speedup. Compared with the better of RailOnly and Permutation, Joint provides additional speedups of $2.125\times$ and $3.295\times$ at 28 MiB and 42 MiB, respectively.

These results show that the preferred execution strategy depends on the traffic pattern. Temporal scheduling is sufficient when the Rails are already balanced, whereas jointly applying spatial and temporal shaping provides substantially larger gains when Rail imbalance and receiver contention coexist.

UltraEP control overhead. We separately measure the UltraEP placement, rerouting, and weight-synchronization overheads, as shown in Table 3. Placement and rerouting remain on the order of tens of microseconds across the evaluated workloads and are small relative to the communication latency.

Table 3: UltraEP control-operation latency in microseconds. These costs are measured separately and are not included in Table 2.

Case	Mode	Placement	Reroute	Weight Sync
N28	Native	84.20	38.51	38.40
N28	RailOnly	87.17	55.02	44.59
N28	Permutation	86.13	36.79	36.03
N28	Joint	84.97	37.84	37.23
N42	Native	84.00	43.76	38.31
N42	RailOnly	85.72	39.70	47.92
N42	Permutation	84.97	39.59	36.45
N42	Joint	85.46	52.76	37.23
R28	Native	82.28	37.58	43.93
R28	RailOnly	84.78	39.94	36.87
R28	Permutation	85.36	38.38	36.87
R28	Joint	85.83	38.82	48.34
R42	Native	84.46	42.57	38.75
R42	RailOnly	85.53	41.55	42.10
R42	Permutation	103.58	63.42	59.94
R42	Joint	87.91	45.16	54.48

Table 4: Calibrated selector anchors. B is average remote payload per GPU per All-to-All operation.

B (MiB)	ρ_g	κ	Selected path
7	1	1	Permutation
7	1	≈ 3	Permutation
14	1	1	Permutation
28	1	1	Native

The weight-synchronization measurements are included only to verify the execution of the UltraEP integration path. The evaluated FC weight tensors contain only 1,024 BF16 elements and therefore do not represent the synchronization cost of full-size MoE expert weights.

A.3 Adaptive Selector Calibration and Validation

Calibration details. The selector uses $x = (\widetilde{B}, \widetilde{\rho}_g, \widetilde{\kappa})$, with

$$\widetilde{B} = \log_2(B/56), \qquad \widetilde{\rho}_g = (\rho_g - 1)/7, \qquad \widetilde{\kappa} = (\kappa - 1)/2.$$

These constants are fixed engineering normalization scales rather than fitted model parameters. The 56-MiB reference corresponds to 32,768 tokens per node under our BF16, hidden-dimension-7,168 workload, and the log transform gives equal spacing to multiplicative changes in payload. For our eight-Rail topology, $7 = R - 1$ scales the range from balanced traffic ($\rho_g = 1$) toward the single-Rail extreme. The factor 2 for κ maps the low-contention workload ($\kappa \simeq 1$) and the high-contention workload ($\kappa \simeq 3$) to approximately 0 and 1. These scales are specific to the calibrated configuration and are re-established when the topology or workload regime changes.

We construct anchors only from repeatable measurements. Both Joint and a candidate alternative must satisfy $p_{90}/p_{10} \leq 1.2$, limiting the accepted run-to-run spread to 20%. We further require $p_{10}(\text{Joint})/p_{90}(\text{candidate}) \geq 1.05$, so that the alternative retains at least a 5% advantage under a conservative cross-quantile comparison. For low-contention development cases, known equivalent-path controls are additionally required to agree within 10%; otherwise the point is rejected as noisy. These filters leave the four anchors in Table 4.

At runtime, an anchor is reused only when the payload is within $1.25\times$ and both normalized shape coordinates are within 0.02. The payload guard permits limited local reuse while keeping the 7-, 14-, and 28-MiB anchor neighborhoods disjoint. The 0.02 shape tolerance is deliberately narrow; under the above normalization it corresponds to deviations of 0.14 in ρ_g and 0.04 in κ. Requests outside

Table 5: Measured validation of the calibrated selector. Speedup ranges include only requests where the selector changes the execution path.

Split	Requests	Beneficial switches	Speedup
Development	12	3/3	1.065–1.234×
Unseen	12	**4/4**	**1.061–1.196×**
Total	24	**7/7**	**1.061–1.234×**

Table 6: Packing overhead on H800. Latency is in ms. Pipelined latencies and reductions relative to Sequential are bold.

MiB/GPU	Prepacked	Sequential	Pipelined	Reduction
10.5	3.200	3.427	**3.301**	**3.68%**
17.5	4.747	5.127	**4.955**	**3.35%**
21	5.503	5.951	**5.760**	**3.21%**
28	7.118	7.692	**7.447**	**3.19%**
35	8.642	9.372	**9.098**	**2.92%**
42	10.210	11.074	**10.756**	**2.87%**

these neighborhoods fall back to Joint. All thresholds and anchors were frozen before evaluating the unseen requests below.

Measured selector results. We evaluate 24 requests on H800: 12 development-regression requests and 12 requests not used to construct the frozen table. Each path uses five warmup iterations, followed by interleaved measurements; we report the median of the first 10 samples. GPU timing covers the complete dispatch–combine interval and takes the maximum across ranks.

Table 5 summarizes the results. The selector switches away from Joint on seven requests, and all seven switches reduce the measured median latency. On the 12 unseen requests, it switches on four cases and obtains 1.061–$1.196\times$ speedup over Joint. Across all seven switched cases, the measured speedup ranges from 1.061–$1.234\times$.

The remaining 17 requests fall back to the same Joint implementation. The Joint and Adaptive columns were nevertheless measured independently, so differences between them on these requests reflect run-to-run variation rather than a path-selection effect. We therefore use only the seven actual path switches to quantify selector benefit.

A.4 Packing Overhead

Permutation-only and Joint require per-wave packing. We compare PREPACKED, SEQUENTIAL, and PIPELINED, where the last overlaps packing of the next wave with communication of the current wave. Table 6 reports dispatch–combine median latency.

As shown in Table 6, pipelining consistently reduces the exposed packing overhead, improving end-to-end latency by 2.87–3.68% over sequential packing. These differences are measured at the full communication-path level rather than as isolated packing-kernel times.

A.5 Static Memory Overhead

RailWave trades redundant expert storage for communication workspace. For D nodes, G local GPUs, per-rank token capacity T, and $C_{\max} = 1024$ reserved channels, the dominant RailBalance workspace is

$$M_{\mathrm{RW}} \approx (D-1)(T + C_{\max})(s_d + s_c) + 4(32C_{\max} + 2C_{\max}DG + DG), \tag{13}$$

where s_d and s_c are the per-record dispatch and return storage. Under our configuration ($D = 4$, $G = 8$, $T = 50{,}176$, $H = 7168$), $s_d = s_c = 2H + 32$, giving approximately 4.11 GiB/GPU of additional workspace. Sequential waves reuse the same workspace rather than allocating one copy per wave.

Table 7: Dominant incremental storage in GiB/GPU. EPLB assumes BF16 expert weights and FP32 replica gradients; RailWave reports the communication workspace above.

r	EPLB inf.	Train, 1-layer grad	Train, all grads	RailWave
1	4.758	4.922	14.273	**4.111**
2	9.516	9.844	28.547	**4.111**
4	19.031	19.688	57.094	**4.111**

EPLB instead incurs persistent storage for redundant experts. If each GPU keeps r additional experts per MoE layer, with L MoE layers, P parameters per expert shard, and w bytes per weight, its dominant inference overhead is

$$\Delta M_{\mathrm{EPLB,inf}} = LrPw. \tag{14}$$

For training, additional replica gradients contribute gG_{live}, where G_{live} depends on how long replica gradients remain resident.

Table 7 gives a static BF16 example using DeepSeek-V3-sized routed experts ($L = 58$, $P = 3 \times 7168 \times 2048$) without expert tensor sharding.

A.6 Replay Workloads and Execution Protocol

Motivation. Megatron-style training (Shoeybi et al., 2020) commonly follows single-program, multiple-data (SPMD) execution: ranks run the same program on different data or model partitions and coordinate through communication and synchronization. This does not require a central controller to track every rank's fine-grained progress. Compute-time variation, rank skew, and network jitter can therefore change communication overlap even when the logical execution order is unchanged. Single-controller runtimes can explicitly coordinate cross-stage tasks (Xhebraj et al., 2024), but centralized coordination is not a defining property of the multiple-program, multiple-data (MPMD) model. We use training-derived layer loads with controlled sending arrangements to cover possible incast conditions, rather than replaying one incidental training timeline.

Sending patterns. For the four-node replay, each layer's measured 128 expert counts are sorted and grouped in sets of 16 to obtain eight load bins indexed by destination-local GPU. Ring sends from node s to $(s+1) \bmod 4$ in one wave, with peak incast degree $[1]$. Moderate and Extreme distribute each source-rank/destination-local-GPU count across the three remote nodes using integer quotient and remainder, preserving each source's total volume and destination-local-GPU histogram. They use the same expanded rank-to-rank demand and three waves, with per-wave peak incast degrees $[2, 1, 2]$ and $[3, 3, 2]$, respectively. Each array entry is the maximum number of source nodes assigned to the same receiver in that wave. Table 8 specifies the node-pair assignments. Extreme is a controlled stress case, not an estimate of how often severe incast occurs in training.

Table 8: Replay schedules. Each tuple lists the destinations of source nodes $(0, 1, 2, 3)$; a dash denotes no wave. Incast degree is the scheduled number of concurrent senders to a receiver, not a measured instantaneous network quantity.

Schedule	Wave 1	Wave 2	Wave 3	Peak incast degree
Ring	$(1, 2, 3, 0)$	–	–	$[1]$
Moderate	$(1, 0, 0, 2)$	$(2, 3, 1, 0)$	$(3, 2, 3, 1)$	$[2, 1, 2]$
Extreme	$(1, 0, 0, 0)$	$(3, 3, 3, 2)$	$(2, 2, 1, 1)$	$[3, 3, 2]$
Permutation	$(1, 2, 3, 0)$	$(2, 3, 0, 1)$	$(3, 0, 1, 2)$	$[1, 1, 1]$

Backend execution and timing. In Moderate and Extreme, Native, Rail-only, and NCCL retain the input three-wave schedule; Permutation-only and Joint reorder the same demand into three one-to-one waves. Each remote node pair appears exactly once, so total volume and wave count are unchanged. Here Native means the DeepEP backend under the specified replay schedule, not a single call with the full layer demand. Ring remains a single wave for these paths. Each sample

times all waves of the layer: dispatch–combine per wave for the DeepEP paths, or forward/return All-to-All for NCCL. CPU planning, index construction, and prepacking are outside this interval.