Understanding and Improving Communication Performance in Multi-node LLM Inference

Prajwal Singhania
Department of Computer Science,
University of Maryland
College Park, MD, USA
prajwal@umd.edu

Siddharth Singh
Department of Computer Science,
University of Maryland
College Park, MD, USA
siddharth9820@gmail.com

Lannie Dalton Hough
Department of Computer Science,
University of Maryland
College Park, MD, USA
ldhough@umd.edu

Akarsh Srivastava
Department of Computer Science,
University of Maryland
College Park, MD, USA
akarsh@umd.edu

Harshitha Menon
Lawrence Livermore National
Laboratory
Livermore, CA, USA
harshitha@llnl.gov

Charles Fredrick Jekel
Lawrence Livermore National
Laboratory
Livermore, CA, USA
jekel1@llnl.gov

Abhinav Bhatele
Department of Computer Science,
University of Maryland
College Park, MD, USA
bhatele@cs.umd.edu

Abstract

As large language models (LLMs) continue to grow in size, distributed inference has become increasingly important. Model-parallel strategies must now efficiently scale not only across multiple GPUs but also across multiple nodes. In this work, we present a detailed performance study of multi-node distributed inference using LLMs on GPU-based supercomputers. We conduct experiments with several state-of-the-art inference engines alongside YALIS, a research-oriented prototype engine designed for controlled experimentation. We analyze the strong-scaling behavior of different model-parallel schemes and identify key bottlenecks. Because all-reduce operations are a common performance bottleneck, we develop NVRAR, a hierarchical all-reduce algorithm based on recursive doubling with NVSHMEM. NVRAR achieves up to 1.9×–3.6× lower latency than NCCL for message sizes between 128 KB and 2 MB on HPE Slingshot and InfiniBand interconnects. Integrated into YALIS, NVRAR achieves up to a 1.72× reduction in end-to-end batch latency for the Llama 3.1 405B model in multi-node decode-heavy workloads using tensor parallelism.

CCS Concepts

• **Computing methodologies** → **Neural networks**; **Distributed algorithms**.

Keywords

Machine Learning, Distributed LLM Inference, Communication

CAIS '26, San Jose, CA, USA

ACM ISBN 979-8-4007-2415-2/2026/05
https://doi.org/10.1145/3786335.3813165

ACM Reference Format:
Prajwal Singhania, Siddharth Singh, Lannie Dalton Hough, Akarsh Srivastava, Harshitha Menon, Charles Fredrick Jekel, and Abhinav Bhatele. 2026. Understanding and Improving Communication Performance in Multi-node LLM Inference. In *ACM Conference on AI and Agentic Systems (CAIS '26), May 26–29, 2026, San Jose, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3786335.3813165

1 Introduction

As large language models (LLMs) grow in size and adoption, inference costs are rising rapidly [6, 15, 21]. Practitioners increasingly rely on larger models, longer sequences, and compute-intensive reasoning to improve output quality [16, 37, 46]. Improving inference performance is therefore critical for reducing energy consumption and operational costs.

With increasing LLM sizes, memory footprints often exceed the capacity of a single GPU, requiring parallel execution across multiple devices. On most clusters, a single node, typically with four to eight GPUs, is insufficient to host large models such as Llama 3.1 405B [9]. To enable inference using such models, inference engines use *model parallelism* [14, 33] schemes, which partition the model parameters across GPUs. While model parallelism for inference within a single node has been extensively studied and optimized [4, 19], inference in *multi-node* settings is comparatively under-explored. In this paper, we systematically study, comparatively evaluate, and optimize multi-node inference workloads.

Multi-node inference introduces challenges such as higher inter-node latencies compared to faster within-node NVLink connections. As a result, parallelization strategies that perform well within a node can experience substantial degradation in multi-node settings due to increased communication overheads. Moreover, the optimal choice of parallelism strategy often depends on the specific inference workload characteristics and the efficiency of underlying communication libraries. Consequently, it remains unclear which

model parallelism schemes are best suited for multi-node inference and how to optimize them further.

In this work, we address the following research questions:

- How do different model parallel schemes (tensor and hybrid parallelism) scale across multiple nodes in distributed environments for specific inference workloads?
- What performance bottlenecks arise in these model parallel schemes under the workloads studied?
- Can we optimize collective communication, which appears as a common performance bottleneck in multi-node inference?

To investigate the questions above, we study the performance of two popular model-parallel schemes: tensor parallelism (TP) and hybrid tensor-pipeline parallelism (HP), in multi-node settings. We evaluate two state-of-the-art inference engines, vLLM [18] and SGLang [52], alongside YALIS, a research-oriented inference engine we develop to facilitate controlled experiments in multi-node HPC environments. We evaluate the performance of these engines on batched inference workloads, and study the scaling behavior of both model-parallel schemes. We identify bottlenecks via detailed performance breakdowns for each of the schemes.

Based on the results of our performance study, we find that workloads that perform better with TP suffer from significant communication overheads from the all-reduce operations. To address this, we propose *NVRAR*: a hierarchical all-reduce implementation built using NVSHMEM [28], and optimized for message sizes occurring in inference workloads. We evaluate NVRAR against NCCL's all-reduce [1] on multiple HPC interconnects, and observe up to 1.9× better performance on HPE Slingshot-11 and 3.6× on InfiniBand networks in the 256 KB to 2 MB message size range. Integrating NVRAR into YALIS and vLLM yields up to a 1.72× improvement in multi-node inference performance for the Llama 3.1 405B model in decode-heavy regimes.

The main contributions of this work are as follows:

- We systematically study the performance of model-parallel inference schemes in multi-node settings, producing detailed performance breakdowns. To facilitate this study, we develop YALIS[1], an inference engine designed for easier experimentation in multi-node HPC environments.
- Based on our performance analysis, we characterize how tensor parallelism compares to pipeline parallelism for different multi-node workloads and across inference phases. We identify bottlenecks in both parallelism schemes.
- To address the communication bottleneck in multi-node TP inference, we develop NVRAR[2], a custom all-reduce implementation optimized for the small-message regime characteristic of decode-heavy workloads. NVRAR delivers up to 1.72× faster multi-node TP inference for Llama 3.1 405B.

2 Background

This section provides background on LLM inference, model parallelization strategies, and the communication primitives they use.

LLM inference consists of two phases: *prefill* and *decode*. In prefill, the model processes all prompt tokens in parallel to generate the first output token and is typically compute-bound due to large matrix multiplications. In decode, it generates subsequent tokens sequentially, becoming memory-bandwidth-bound because of smaller matrix multiplications and frequent parameter/KV-cache accesses.

2.1 Model Parallelism for Inference

LLMs that exceed the memory capacity of a single GPU require distributing model parameters and computations across multiple GPUs. This is broadly referred to as *model parallelism*, which can be implemented in several ways. In *pipeline parallelism (PP)*, contiguous groups of layers are assigned to P processing units (pipeline stages), forming a sequential dependency chain with point-to-point communications. It achieves high utilization by splitting a batch of prompts into pipelined micro-batches. In *tensor parallelism (TP)*, the computation of each layer is partitioned across GPUs by splitting the underlying matrix multiplications. TP has no sequential dependency between GPUs, but aggregation of partial results incurs high communication overheads due to per-layer all-reduce operations.

2.2 Algorithms for All-reduce

NCCL [1] is the default communication library for AI workloads on NVIDIA GPUs and primarily implements two all-reduce algorithms: *Ring* and *Tree* [13]. Other variants, such as *CollNet*, depend on specialized DGX hardware and are out of scope for this study. We model the performance of Ring and Tree all-reduce using the α-β communication model [12]. Consider a system with N nodes, each containing G GPUs. The inter-node network has latency α_{inter} and bandwidth β_{inter}, while the intra-node interconnect has latency α_{intra} and bandwidth β_{intra}, where $\alpha_{\text{intra}} < \alpha_{\text{inter}}$ and $\beta_{\text{intra}} > \beta_{\text{inter}}$. Let M denote the input message of size $|M|$ bytes.

Ring all-reduce. NCCL's Ring all-reduce performs a reduce-scatter followed by an all-gather over a flat ring topology where all links are active each step. Inter-node links dominate the cost, and the communication time is modeled as:

$$T_{\text{ring}} = 2(NG-1)\alpha_{\text{inter}} + 2\frac{NG-1}{NG}\left(\frac{|M|}{\beta_{\text{inter}}}\right) \tag{1}$$

Tree all-reduce. The Tree all-reduce performs a reduction followed by a broadcast using a double binary tree topology [27] for inter-node communication and a simple intra-node chain. The communication time is modeled as:

$$T_{\text{tree}} \approx 2(G-1)\,\alpha_{\text{intra}} + 2\log_2(N)\,\alpha_{\text{inter}} + 2\frac{N-1}{N}\left(\frac{|M|}{\beta_{\text{inter}}}\right) \tag{2}$$

For simplicity, we consider only the inter-node bandwidth term in the above expression ($\beta_{\text{intra}} \gg \beta_{\text{inter}}$).

NVSHMEM. An OpenSHMEM-based [7] communication library from NVIDIA, providing host and device APIs for one-sided put/get and collective operations over NVLink, Slingshot, and InfiniBand. It enables implementation of GPU-initiated communication kernels.

3 Studying Performance of Multi-node Inference

This section presents a performance study of multi-node LLM inference. Our objective is to evaluate different model-parallelism schemes, understand their scaling behavior, and identify bottlenecks for batched inference workloads. We first introduce YALIS, a

[1] https://github.com/axonn-ai/yalis
[2] https://github.com/hpcgroup/nvrar

prototype inference engine built for controlled performance studies. We then detail our benchmarking methodology and present our experimental results, discussing the performance of YALIS and existing state-of-the-art inference engines in multi-node settings.

3.1 YALIS: Yet Another LLM Inference System

YALIS is an open-source inference engine built as a research vehicle to study multi-node LLM inference. It is intended to be performant, easy to instrument, and more amenable to Slurm-based environments. These properties allow for detailed analysis of inference performance on HPC systems. Its design is centered around three key components: (1) a unified model definition layer, adapted from LitGPT [20], providing compatibility with a wide range of model architectures; (2) an execution layer utilizing Torch Compile [22] for kernel fusion and optimization, and CUDA Graphs [26] for minimal kernel-launch overheads; and (3) tensor model parallelism implemented via AxoNN [34–36], both within and across nodes.

3.2 Benchmarking Methodology

Table 1: Details of the HPC systems used in our experiments.

System	GPU	GPUs/Node	Interconnect
Perlmutter	A100 (40/80 GB)	4	Intra-Node: 3rd gen NVLink Inter-Node: Slingshot-11
Vista	GH200 (96 GB)	1	Inter-Node: InfiniBand

Hardware and Models. Our scaling experiments use the Perlmutter system [23] (Table 1) (80 GB nodes unless otherwise specified). We evaluate two dense LLMs — Llama 3.1 70B (Instruct) and 405B (Instruct) [9] — in bf16 precision. While we evaluate a single model family, its architecture and compute, memory, and communication patterns are representative of other dense LLMs.

Table 2: Details of workloads evaluated in our experiments.

Workload	Prompt Length	Decode Length	NumPrompts (#P)
Prefill-heavy	2363	128	8, 32
Decode-heavy	1426	3072	8, 32

Workloads and Metrics. Table 2 lists the workload configurations used in our experiments. We define *NumPrompts* (#P) as the number of prompts provided to the inference engine in a single user batch. For brevity, we present a subset of results in the main text, with additional results in Appendix A.

In *batched inference* workloads, the engine processes one batch of prompts to completion before we submit the next batch. This mirrors real-world settings such as synchronous GRPO [11, 32] and helps isolate GPU execution performance from scheduler effects. We report the total time-to-completion for a single batch of prompts.

In our strong scaling experiments (fixed workload across GPUs), the 70B model is scaled from four GPUs (single node) to 32 GPUs (eight nodes), and the 405B model is scaled from 16 GPUs (four nodes) to 128 GPUs (32 nodes). Each run includes two warm-up and up to three timed generations. We repeat each run three times and report average performance. For performance breakdowns, we use one run with two warm-up and one profiled generation.

Table 3: Parallelism schemes and inference engines.

Parallelism	Intra-Node	Inter-Node	Engines
Tensor Parallelism (TP)	TP	TP	YALIS, vLLM V1 (v0.11.0), SGLang (v0.5.1)
Hybrid Parallelism (HP)	TP	PP	vLLM V0 (v0.10.0), SGLang (v0.5.1)

Software Stack and Parallelism Schemes. Table 3 lists the inference engines we use in our experiments for different parallelism schemes. We use PyTorch 2.8 [30] and CUDA 12.9 for all experiments. For vLLM, we use the V0 engine for HP because we observed persistent hangs with the V1 engine when using Ray-based PP on Slurm-based systems [31]. For performance breakdowns, we use Nsight Systems [24] to collect traces and Pipit [5] to analyze them.

3.3 Scaling Multi-node LLM Inference

Figures 1 and 2 report the time-to-completion for a batch of prompts across all engines for Llama 3.1 70B and 405B on Perlmutter, respectively. From left to right, the workloads transition from compute-bound to increasingly memory-bound regimes.

We first observe that YALIS (orange line) achieves comparable performance to state-of-the-art engines, particularly for memory-bound workloads. For the 70B model, YALIS is within 5–16% of vLLM (TP) at eight GPUs and beyond. For the 405B model, it is within 8% for all GPU counts. The only noticeable deviation occurs for the 70B model's prefill-heavy workload at 16 GPUs. Crucially, YALIS exhibits scaling trends consistent with other engines, validating its suitability as a research vehicle for studying multi-node LLM inference. The missing data points correspond to OOM errors.

Across all models and engines, both TP and HP exhibit poor strong scaling, where the time to solution does not scale inversely with GPU count. Focusing on the 70B model (Figure 1), vLLM (TP) (green line) latencies decrease from four GPUs (single node) to eight GPUs (two nodes), with more noticeable improvements for the decode-heavy workload (right-most plot). However, after 16 GPUs, the latency remains almost constant or increases with GPU count. This trend is consistent for TP, across all engines and models.

When using HP, we observe a different trend. In the prefill-heavy regime for the 70B model (Figure 1), vLLM (HP) (black line) latencies remain nearly constant with a smaller number of prompts (middle plot), but decrease (up to 16 GPUs) with a larger number of prompts (left plot). For the 405B model, latencies decrease initially for both small and large numbers of prompts, before increasing or flattening out. SGLang (HP) (pink line) exhibits a similar trend. For decode-heavy workloads, however, HP latencies increase significantly with increasing GPU count for both vLLM and SGLang.

Comparing the two schemes, HP outperforms TP for the most compute-bound and prefill-heavy workload (Figure 1, left), but TP starts to outperform HP as workloads become more memory-bound and decode-heavy. This holds across engines (Figure 1, middle and right). Similar trends are observed for the 405B model (Figure 2).

Observation 1: For the workloads studied, TP and HP do not scale ideally. HP is advantageous in compute-bound regimes, whereas TP is better for memory-bound and decode-heavy cases.

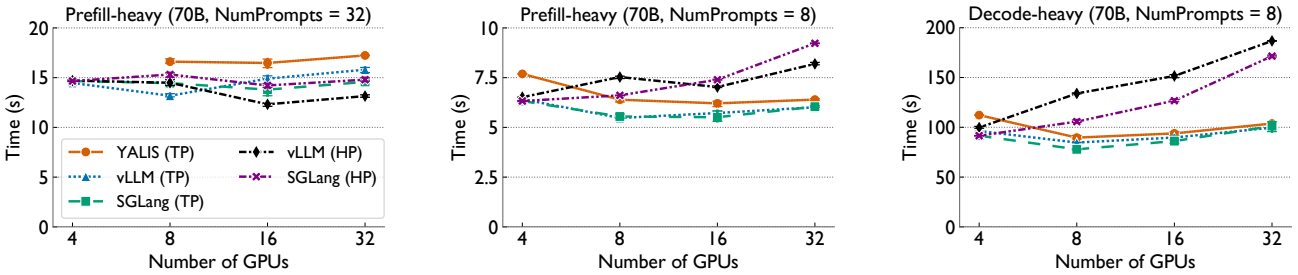

Figure 1: Strong scaling performance of different inference engines on Perlmutter for Llama 3.1 70B Instruct. The Y-axis denotes the end-to-end latency per batch in seconds, and the X-axis denotes the number of GPUs.

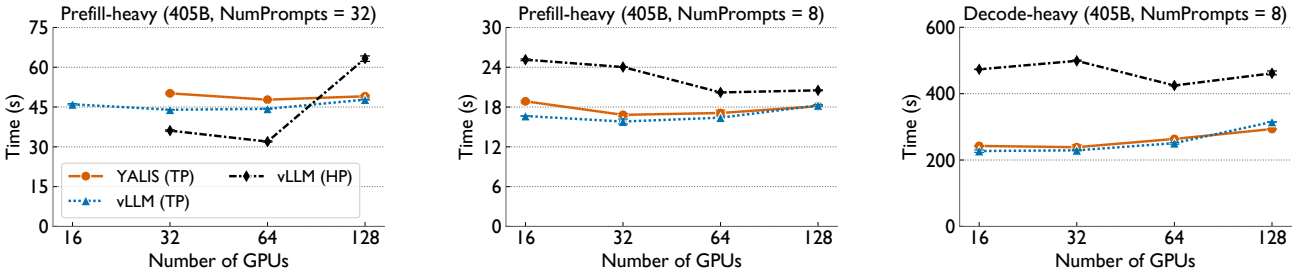

Figure 2: Strong scaling performance of different inference engines on Perlmutter for Llama 3.1 405B Instruct. The Y-axis denotes the end-to-end latency per batch in seconds and the X-axis denotes the number of GPUs.

3.4 Identifying Performance Bottlenecks

To better understand the scaling behaviors of TP and HP across prefill- and decode-heavy workloads, we analyze the performance breakdowns of YALIS (TP) and vLLM (HP) on eight and 16 GPUs for the 70B model (Figure 3). We decompose the total time into four components: *Matmul* (time spent in matrix multiplications), *Other Comp.* (time spent in other computations), *Comm.* (time spent in communication), and *Idle* (per-GPU idle time). We report per-GPU breakdowns rather than aggregating across the critical path.

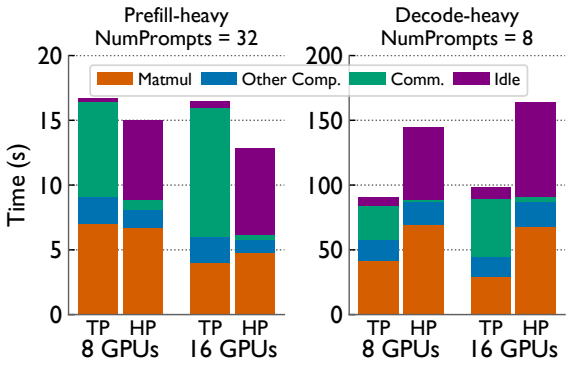

Figure 3: Performance breakdown of TP (using YALIS) and HP (using vLLM) for the prefill-heavy and decode-heavy workloads on Perlmutter for the 70B Llama model.

For the prefill-heavy workload (Figure 3, left), both YALIS (TP) and vLLM (HP) reduce computation time going from eight to 16 GPUs, with vLLM (HP) achieving lower overall latency due to reduced communication overhead. However, vLLM (HP) exhibits unexpectedly high GPU idle time. We hypothesize that this behavior is due to pipeline bubbles in batched inference caused by imbalanced prefill and decode stage times. Continuous batching [49] can mitigate this, but we leave detailed investigation to future work.

Table 4: Synthetic benchmarks modeling Prefill-GEMM (M=32768, N=8192, K=57344) and Decode-GEMM (M=32, N=8192, K=57344) matrix multiplications in the MLP layer of the 70B Llama model. $M/2$ corresponds to HP micro-batching and $K/2$ corresponds to TP.

Workload	Baseline (M,N,K)	HP ($M/2$,N,K)	TP (M,N,$K/2$)
Prefill-GEMM	108.033 ms	53.824 ms	53.896 ms
Decode-GEMM	0.614 ms	0.574 ms	0.359 ms

For decode-heavy workloads, HP fails to reduce the time spent in matrix multiplications, unlike TP. This partially explains why, despite lower communication costs, HP does not scale as well for such workloads. To isolate this behavior, we run a synthetic GEMM benchmark using two representative matrix sizes: Prefill-GEMM (M=32768, N=8192, K=57344) and Decode-GEMM (M=32, N=8192, K=57344). The former models large-M prefill matmuls (batch size $\times$ prompt length), while the latter models small-M decode matmuls (batch size $\times$ 1). Table 4 reports the runtime of both when either M is halved (micro-batching in the PP phase of HP) or K is halved (TP). For Prefill-GEMM, halving either M or K nearly halves the runtime. For Decode-GEMM, however, halving K reduces the runtime substantially, whereas halving M yields only a marginal reduction. This

behavior arises due to tiling in GEMM kernels, where decreasing M below the tile size yields no speedup. While TP outperforms HP for these workloads, it still incurs significant communication overhead. Figure 3 (right) highlights that the communication time in YALIS (TP) increases by ~1.6×, offsetting the gains from reduced computation time when going from eight to 16 GPUs.

Observation 2: For prefill-heavy workloads, both TP and PP reduce computation time, with PP achieving lower overall latency due to its reduced communication overhead. For decode-heavy workloads, PP does not reduce matrix multiplication time, while TP suffers from significant communication overhead.

3.5 Communication Issues in Tensor Parallelism

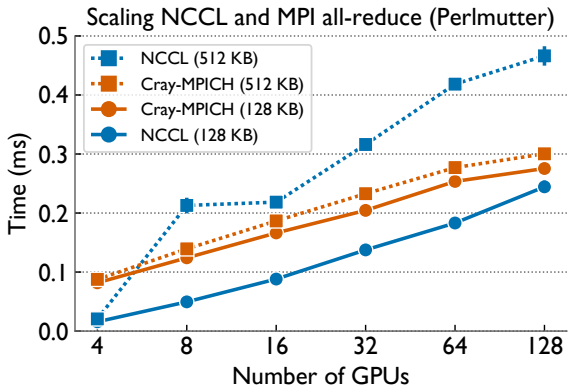

Figure 4: Scaling performance of NCCL and MPI all-reduce for a range of message sizes on Perlmutter.

The primary communication collective in TP is all-reduce. In the decode-heavy regime, it is dominated by small messages of size $B \times H$, where B is the batch size and H the hidden dimension. For the 70B model with B=8 and H=8192, this message size is 128 KB. Across our workloads, message sizes range from 128 KB to 1 MB.

To further analyze communication bottlenecks, we benchmark NCCL all-reduce against GPU-aware Cray-MPICH on Perlmutter (40 GB nodes), focusing on small messages. We run the OSU benchmark [42] and `nccl-tests` [25], and report average all-reduce time over 10 runs (200 warm-up and 10,000 timed iterations) in Figure 4. We observe that NCCL all-reduce is substantially faster within a node, but scales poorly across nodes compared to MPI. For 512 KB–1 MB messages, NCCL is 1.5–2× slower than MPI, with latency growing faster with message size at any given scale. While Cray-MPICH's implementation is proprietary, the open-source MPICH [10] library typically employs a latency-optimal recursive-doubling algorithm [40] for these regimes, which can explain the performance gap.

Observation 3: For small message sizes, typical in the decode phase, NCCL all-reduce exhibits poor scaling across nodes and can at times be slower than MPI.

Algorithm 1: NVRAR

Input: Message M; GPUs/node G; number of nodes N; chunk size C_s; sequence number seq; GPU rank (within node) r_g; node rank r_n; pre-allocated send/receive buffers B_{send}, B_{recv}
Output: M reduced in-place

1 **Function** NVRAR($M, G, N, C_s, seq, r_g, r_n$):
2 $M' \leftarrow$ REDUCE-SCATTER$_{intra}(M, G)$;
3 $seq \leftarrow seq + 1$;
4 **for** $i = 0$ **to** $\log_2 N - 1$ **do**
5 $peer_i \leftarrow (r_n \oplus 2^i, r_g)$;
6 WAIT($peer_i$, seq) // Synchronize Sequence Number
7 **end**
8 $B_{send}[0] \leftarrow$ PACKDATAANDSEQNUM(M', seq);
9 $B_{out} \leftarrow$ RD$_{\text{inter}}([B_{send}, B_{recv}], N, C_s, seq, r_g, r_n)$;
10 $M' \leftarrow$ UNPACKDATAANDSEQNUM(B_{out}, seq);
11 $M \leftarrow$ ALL-GATHER$_{intra}(M', G)$;
12 **Function** RD$_{\text{inter}}(B_{send}, B_{recv}, N, C_s, seq, r_g, r_n)$:
13 $Q \leftarrow \lceil |B_{send}[0]| / C_s \rceil$ // Number of chunks
14 **for** $\ell = 0$ **to** $\log_2 N - 1$ **do**
15 $peer_\ell \leftarrow (r_n \oplus 2^\ell, r_g)$;
16 **for** $q = 0$ **to** $Q - 1$ **do**
17 $src_q \leftarrow B_{send}[\ell][q]$; $dst_q \leftarrow B_{recv}[\ell][q]$;
18 NON-BLOCKING-PUT: $src_q \rightarrow peer_\ell$'s dst_q;
19 Wait until $flag(dst_q) == seq$ // Wait for data
20 $B_{send}[\ell + 1][q] \leftarrow dst_q + src_q$;
21 **end**
22 **end**
23 **return** $B_{send}[\ell]$;

4 Optimized Multi-node All-reduce

We establish in the previous section that TP is generally more performant for decode-heavy workloads. Next, we focus on optimizing its communication bottlenecks. One potential approach to address NCCL's all-reduce performance issues is to use MPI wherever it is faster than NCCL. However, standard MPI implementations are ill-suited for inference workloads due to the lack of CUDA Graph support and sub-optimal intra-node NVLink communication. As an alternative to NCCL and MPI, we develop NVRAR: an NVSHMEM-based hierarchical all-reduce implementation that uses the recursive-doubling algorithm and is optimized for small-message inter-node communication.

4.1 Three-Phase Hierarchical All-reduce Design

NVRAR (Algorithm 1) has three phases: (1) an intra-node reduce-scatter, (2) an inter-node recursive-doubling all-reduce, and (3) an intra-node all-gather. Figure 5 illustrates this design for an N-node system with G GPUs/node.

Reduce-scatter Phase: In the first phase (Line 2 of Algorithm 1), GPUs within a node perform a local reduce-scatter operation. For input message M of size $|M|$ bytes, each GPU holds $\frac{|M|}{G}$ bytes of data reduced within each node at the conclusion of this phase. We implement this using the `nvshmemx_TYPENAME_sum_reducescatter` host-side API, which internally calls NCCL reduce-scatter.

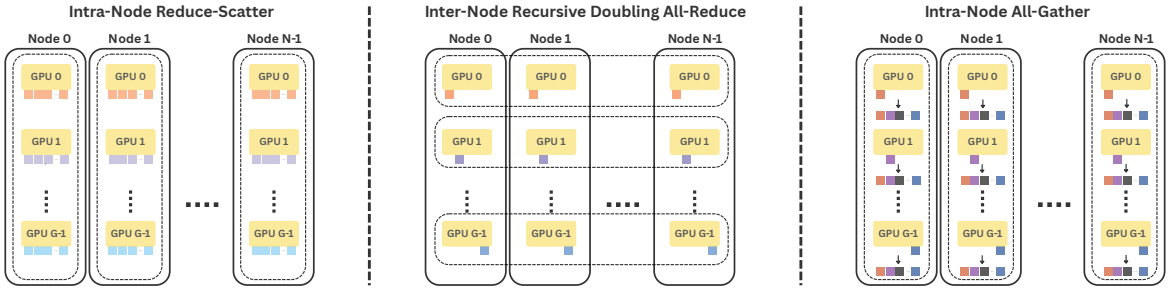

Figure 5: Three-phase NVRAR design: (1) intra-node reduce-scatter, (2) inter-node all-reduce, (3) intra-node all-gather.

Inter-Node Recursive-Doubling Phase: In the second phase (Line 9 of Algorithm 1), corresponding GPUs with the same local ID on each node perform an all-reduce. Each node is identified by its ID $r_n \in [0, N-1]$, and each GPU within a node by its local ID $r_g \in [0, G-1]$, so each GPU is uniquely identified by the pair (r_n, r_g). This phase takes $\log_2 N$ steps. At each step $0 \leq i < \log_2 N$, GPU (r_n, r_g) exchanges data with its 2^i-th logical peer, $(r_n \oplus 2^i, r_g)$, where $\oplus$ denotes bitwise XOR. Thus, GPUs with the same local ID communicate across nodes. Upon receiving data, each GPU performs a local reduction with the received buffer before proceeding to the next step. After all $\log_2 N$ steps, each GPU holds $\frac{|M|}{G}$ bytes of the globally reduced data. We implement this phase with a custom NVSHMEM kernel using non-blocking `put_nbi`-based RMA primitives.

All-gather Phase: In the third and final phase (Line 11 of Algorithm 1), the GPUs within a node perform a local all-gather operation to combine their $\frac{|M|}{G}$ fraction of the globally reduced data into a single tensor. Similar to the reduce-scatter phase, we implement this using NVSHMEM's host API. After completion, every GPU holds the full globally reduced tensor, completing the all-reduce.

4.2 Performance Optimizations

The inter-node phase of NVRAR contributes most to the overall all-reduce runtime. Apart from choosing the algorithmically optimal recursive-doubling approach, we make three key optimizations for increased efficiency and lower latency: (1) chunked non-blocking communication, (2) fused data-flag payloads for per-step synchronization, and (3) sequence-number-based global synchronization.

4.2.1 Chunked Non-Blocking Communication. Efficient utilization of GPU SMs and concurrent progress of data transfers and reductions across thread blocks are crucial for all-reduce performance. To achieve this, we partition the message into disjoint data blocks, processed independently by B_s thread blocks. Each thread block further subdivides its data into chunks of size C_s bytes (Lines 15–21 in Algorithm 1). Each chunk is transmitted to the corresponding peer GPU using a non-blocking, block-cooperative NVSHMEM primitive. Upon receiving a peer's chunk, the thread block performs a local reduction and advances to the next chunk.

This design allows different thread blocks to progress through different stages of the all-reduce concurrently, creating coarse-grained computation-communication overlap across SMs. Non-blocking communication allows asynchronous sending and waiting for data, while per-block chunking offers tunable control over the granularity of network injection. We tune B_s and C_s once for a given message size and node count, as performance is sensitive to these hyperparameters (Appendix C.1).

To further enable asynchronous progress across inter-node steps, we allocate per-step send and receive buffers. This allows sending data to the next peer before the receiving peer has completed its previous step. The extra memory overhead is negligible for small messages and logarithmic recursion depth.

4.2.2 Fused Payloads for Step Synchronization. At each recursive-doubling step, peers need to synchronize to ensure the completion of remote data receipt before performing local reduction. A straightforward approach is to use NVSHMEM's explicit signaling primitives — `put_with_signal` and `wait_until`. However, these primitives introduce non-trivial latency overheads, particularly on Slingshot networks. The root cause lies in NVSHMEM's current libfabric implementation, where `put_with_signal` relies on software fences instead of Slingshot's hardware fences and message ordering capabilities.

To avoid explicit signaling, we adopt NCCL's low-latency `LL` protocol design, fusing data and synchronization flags into a single 8 B payload (4 B data + 4 B flag). This granularity ensures atomic and ordered delivery of each data word and its flag, on both Slingshot and InfiniBand. Fused payloads also allow reductions to begin immediately upon receipt (at a warp level), enabling fine-grained progress and synchronization without extra communication overhead.

4.2.3 Sequence-Number-Based Global Synchronization. When all-reduce operations are issued in succession, previous operations must complete before the intermediate buffers can safely be reused. NVSHMEM provides `quiet` and `fence` primitives to achieve this, but they add significant latency overheads. Instead, we assign each all-reduce operation a unique sequence number (Line 2 of Algorithm 1). Each rank waits for its peers to reach the same sequence number (Lines 4–6 of Algorithm 1) before sending data in the inter-node communication phase.

Crucially, each rank synchronizes only with its peers, rather than with all ranks through a global barrier. This synchronization occurs at the beginning of the all-reduce operation, rather than at the end, allowing each rank to use the all-reduced data immediately after its operation completes. Waiting for peer ranks to finish is deferred until the next all-reduce is issued. We implement this using NVSHMEM's atomics.

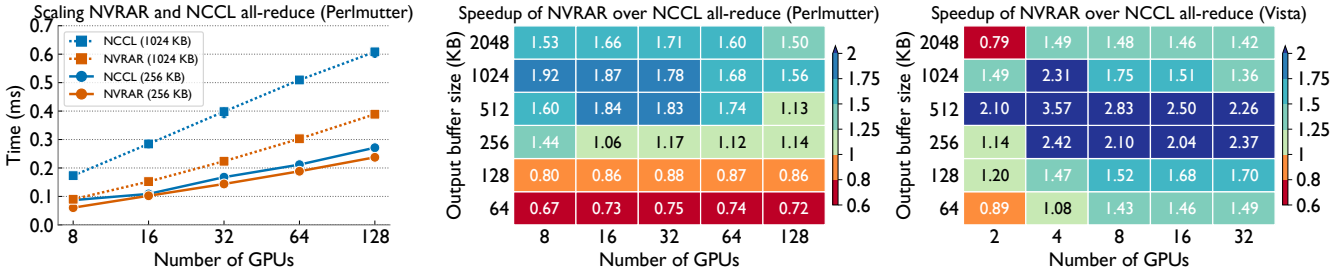

Figure 6: Performance comparison of NVRAR and NCCL all-reduce. (Left) Scaling of NVRAR and NCCL all-reduce for 256 KB and 1024 KB messages across GPU counts on Perlmutter. (Middle, Right) Speedup of NVRAR over NCCL across message sizes and GPU counts on Perlmutter (A100, Slingshot-11) and Vista (GH200, InfiniBand).

4.3 Performance Model for NVRAR

To analyze NVRAR's performance, we model its communication time using the α-β model and notations from Section 2.

Reduce-scatter Phase. Within a node, NVRAR uses NCCL reduce-scatter (Ring), modeled as:

$$T_{\text{RS}} = (G-1)\alpha_{\text{intra}} + \frac{G-1}{G}\left(\frac{|M|}{\beta_{\text{intra}}}\right) \tag{3}$$

Inter-Node Recursive-Doubling Phase. The inter-node phase proceeds in $\log_2(N)$ steps across N nodes, with a message size of $|M|/G$. Packing the data and flag leads to a $1 < \eta < 2$ factor increase in the message size. Each step requires a single exchange between peers and thus, the communication time is:

$$T_{\text{RD}} = \log_2(N)\alpha_{\text{inter}} + \frac{N-1}{N}\left(\frac{\eta|M|}{G\,\beta_{\text{inter}}}\right) \tag{4}$$

All-gather Phase. Finally, the results are aggregated within each node by NCCL's all-gather (Ring), modeled as:

$$T_{\text{AG}} = (G-1)\alpha_{\text{intra}} + \frac{G-1}{G}\left(\frac{|M|}{\beta_{\text{intra}}}\right) \tag{5}$$

Total Communication Time. Combining the three phases, we get the total communication time for NVRAR as:

$$T_{\text{NVRAR}} = 2(G-1)\alpha_{\text{intra}} + \log_2(N)\alpha_{\text{inter}} + \frac{|M|}{G}\left[\frac{2(G-1)}{\beta_{\text{intra}}} + \frac{(N-1)\eta}{N\,\beta_{\text{inter}}}\right] \tag{6}$$

For small messages, the communication time is latency-dominated and the bandwidth terms are negligible. In this regime, NVRAR scales as $O(\log_2 N)$ with the number of nodes, whereas Ring all-reduce (1) scales linearly. NVRAR therefore matches the logarithmic scaling of Tree all-reduce (2), but has a lower inter-node latency coefficient because each recursive-doubling step requires only a single peer exchange.

5 Results and Discussion

This section presents a detailed performance evaluation of NVRAR against NCCL, both as an independent collective primitive and within the context of tensor-parallel inference workloads.

Additional Setup Details. We use Perlmutter and Vista (Table 1) for our evaluations. To isolate collective performance, we run a microbenchmark that executes NCCL all-reduce and NVRAR, each within a CUDA Graph, for 100 consecutive iterations. It replays the captured graph 1,000 times (200 warm-up iterations), and we report the average all-reduce time per collective call per iteration. CUDA Graphs help mimic inference workloads more accurately. We use NCCL 2.27.3 and PyTorch 2.8 for all experiments.

For end-to-end evaluation, we integrate NVRAR into YALIS and vLLM. We run the decode-heavy workload (Table 2) on both engines and compare TP performance using NVRAR against NCCL all-reduce, reporting the relative speedup in end-to-end batch time. We also evaluate a realistic workload trace with 1,000 prompts sampled from BurstGPT [45], using vLLM's benchmarking framework [43]. The trace contains a mix of prefill- and decode-heavy requests (Appendix C.4.2). For this setting, we report the output throughput of NVRAR-based TP, NCCL-based TP, and HP deployments. We use vLLM V1 for HP here because we do not observe the hangs encountered in batched inference evaluations.

5.1 Comparing NVRAR and NCCL All-reduce

Figure 6 (left) reports the all-reduce microbenchmark performance for 256 KB and 1024 KB messages on Perlmutter. NVRAR (orange line) scales linearly with GPU count on a logarithmic X-axis, consistent with our theoretical model (6). For 1024 KB messages on Perlmutter, NCCL (blue line) selects the Tree algorithm (LL protocol) at all GPU counts and scales logarithmically as well. This setting allows us to directly compare the two algorithms and attribute NVRAR's better performance to its lower latency coefficients, consistent with our model. For 256 KB messages, however, NCCL switches from the Ring to the Tree algorithm beyond 16 GPUs, which complicates a direct theoretical comparison. Empirically, NVRAR outperforms NCCL at most GPU counts for both message sizes. We observe similar trends on Vista (Appendix C.3).

Figure 6 (middle and right) also presents the relative speedup of NVRAR over NCCL all-reduce for a range of message sizes and GPU counts. On Perlmutter (middle), NVRAR is slower than NCCL for 64 KB and 128 KB messages, partly due to kernel launch overheads from its three-phase design. We also find that the microbenchmark can slightly underestimate speedup compared to real-world workloads with interleaved communication and computation (Appendix B). Between 256 KB and 1 MB, NVRAR achieves significant speedups over NCCL (1.06×–1.92×). On Vista (right), we observe even higher speedups. Beyond four GPUs, for 64 KB and 128 KB messages, NVRAR outperforms NCCL by 1.08×–1.70×. Between

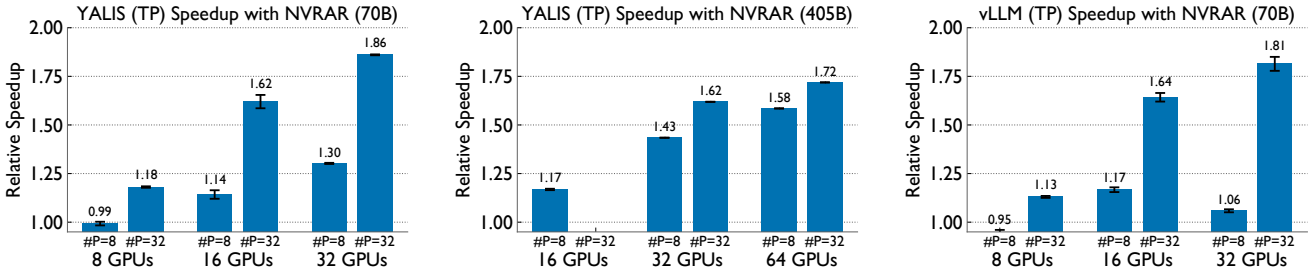

Figure 7: Relative speedup of YALIS (TP) and vLLM (TP) using NVRAR over NCCL all-reduce for the decode-heavy workload on Perlmutter, across different models and NumPrompts (#P).

256 KB and 1 MB, NVRAR can achieve up to 3.5× lower latency than NCCL. We attribute these higher speedups to the architecture of Vista, where each node has one GPU. As a result, NVRAR only executes the inter-node recursive-doubling phase and has lower kernel launch overheads.

To ensure that the reported speedups are not an artifact of suboptimal algorithm selection in NCCL, we fix the all-reduce algorithm to Tree and Ring individually and confirm that NVRAR's gains persist (Appendix C.3). We further verify that results hold across NCCL versions by comparing NCCL 2.27.3 (used in our evaluation) against NCCL 2.28.9 (shipped with the latest PyTorch 2.11). Both versions perform similarly, and NVRAR's speedups hold against both (Appendix C.3.3).

5.2 Improving Multi-node TP Inference

We now present the end-to-end performance evaluation when using NVRAR with YALIS and vLLM in multi-node inference settings.

5.2.1 Batched Inference Performance. Figure 7 (left and middle) reports the relative speedup of YALIS (TP) with NVRAR over NCCL all-reduce for the 70B and 405B models on Perlmutter under the decode-heavy workload. For the 70B model (left), NVRAR accelerates YALIS (TP) by 1.3× for NumPrompts(#P)=8 on 32 GPUs. For this configuration, the all-reduce message size is 128 KB, where our standalone microbenchmark reports slowdowns with NVRAR (Figure 6). We attribute this to the microbenchmark launching all-reduces back-to-back without interleaved computation, unlike real workloads. As a result, NVRAR's deferred peer synchronization (Section 4.2.3) triggers immediately at the start of the next all-reduce, without any computation to hide it. When we run the microbenchmark with interleaved computation, the trends match end-to-end results (Appendix B). For #P=32 on 32 GPUs (message size 512 KB), NVRAR-based TP achieves a 1.86× speedup over NCCL-based TP. For the 405B model (middle plot), speedups range from 1.17×–1.72×, aided by more favorable message sizes (256 KB and 1024 KB) compared to the 70B model. We observe similar speedups on Vista (Appendix C.4). With vLLM (TP), NVRAR delivers up to 1.81× speedup for the 70B model (Figure 7 right), demonstrating that the gains generalize beyond YALIS.

5.2.2 Performance Breakdown. To better understand the performance gains observed with NVRAR, we analyze the breakdown of end-to-end time for YALIS (TP) using NVRAR and NCCL all-reduce on 16 GPUs of Perlmutter for the 70B model in Figure 8. For both #P values, NVRAR leads to a lower communication time than NCCL all-reduce. The decrease is more pronounced for #P=32 compared to #P=8, due to the more favorable message size (512 KB vs. 128 KB). Idle time is marginally higher for NVRAR, but not enough to offset the overall performance gains. We plan to investigate and reduce idle time in future versions of NVRAR.

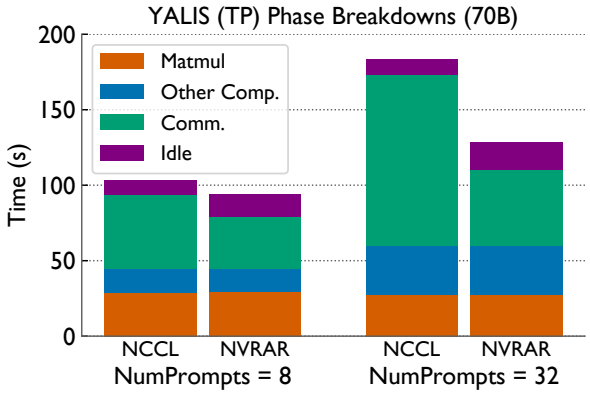

Figure 8: Per-phase time breakdown of YALIS (TP) using NVRAR and NCCL all-reduce for the decode-heavy workload on 16 GPUs of Perlmutter.

5.2.3 Trace-based Performance Evaluation. Finally, we evaluate NVRAR with vLLM in a serving setup, with a trace sampled from BurstGPT. Figure 9 reports the output throughput of NVRAR-based TP, NCCL-based TP, and HP deployments under two maximum request concurrency settings (C=32 and 256). Compared to NCCL-based TP, NVRAR delivers 1.12×–1.36× higher throughput at C=32. At C=256, the improvement decreases to 1.04×–1.09×, because NVRAR primarily benefits small messages (128 KB–4 MB) and higher concurrency pushes the decode message sizes towards the upper bound of this range. Compared to HP, NVRAR-based TP achieves 1.15×–1.20× higher throughput at C=256. We observe smaller gains at C=32 in this case. This partly stems from the mixed prefill-decode batching in vLLM. At lower concurrency, prefills are dispersed, increasing mixed prefill-decode batches and all-reduce message size per batch. At higher concurrency, prefills finish earlier as more requests are processed in parallel, leading to more decode-only batches, where NVRAR performs best. Overall, NVRAR-based TP outperforms NCCL-based TP by 1.04×–1.36× across both settings,

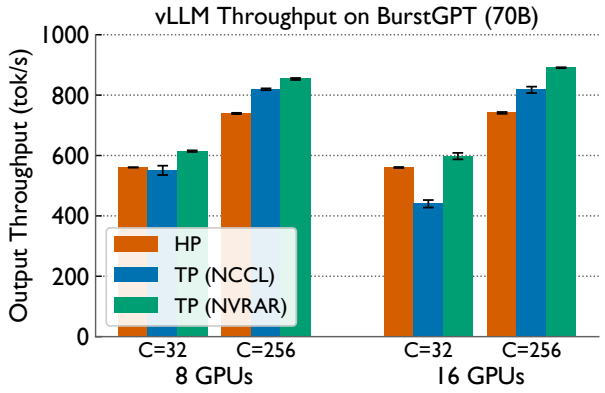

Figure 9: Output throughput on BurstGPT trace when serving the 70B model on Perlmutter with vLLM. We compare NCCL-based TP, NVRAR-based TP, and HP under two maximum request concurrency settings (C=32 and C=256).

and outperforms the best NCCL-based deployment by 1.04×–1.10×, validating its usefulness for real-world workloads. For decode-heavy traces, the gains are more pronounced (Appendix C.4.3).

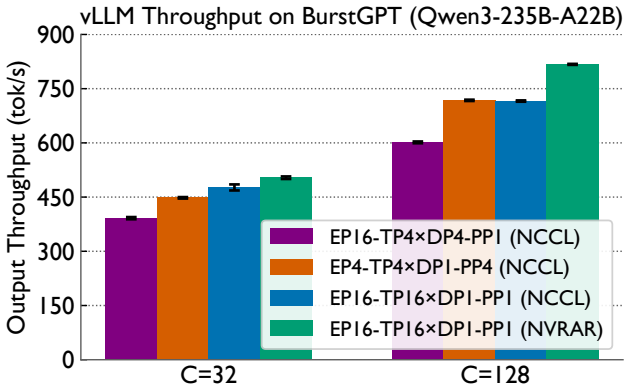

Figure 10: Output throughput on BurstGPT trace when serving Qwen3-235B-A22B on 16 GPUs of Perlmutter with vLLM. We compare expert-parallel deployments: EP partitions the MoE layers, TP×DP partitions the non-MoE layers, and PP partitions the model end-to-end. All configurations use NCCL except the last, which uses NVRAR. We evaluate two maximum request concurrency settings (C=32 and C=128).

5.2.4 Applicability to MoE Models. Mixture-of-experts (MoE) models increasingly combine expert parallelism (EP) in the MoE layers with TP in the non-MoE layers. Since NVRAR targets TP all-reduce communication, it is orthogonal to EP optimizations and can accelerate such MoE deployments. To validate this, we evaluate vLLM with NVRAR on a multi-node deployment of Qwen3-235B-A22B on 16 GPUs of Perlmutter and the BurstGPT trace. Figure 10 compares four different parallelism configurations, all using NCCL except the last, which uses NVRAR for the TP all-reduce. The TP16-EP16 configuration with NVRAR achieves the highest throughput at both concurrency settings. It achieves up to ~1.14× higher throughput than the best NCCL-based configuration at C=128. These results confirm that NVRAR generalizes beyond dense models and provides meaningful gains in MoE deployments where TP remains a critical communication bottleneck.

Our evaluation demonstrates that NVRAR achieves strong performance improvements over NCCL all-reduce for small message sizes, and when integrated into inference engines, it leads to significant gains in multi-node TP performance.

6 Related Work

Model Parallel Performance Studies. Several prior works investigate model-parallel schemes for inference [3, 38, 39, 48, 50, 53], in tandem with other system optimizations. However, these evaluations are often limited to single-node or two-node settings and smaller models. vLLM benchmarked Llama 3.1 405B performance [44] on up to 16 GPUs (two nodes), comparing InfiniBand and non-InfiniBand networks, exposing weak TP performance on the latter. In contrast, our work presents a detailed systematic performance study for large models in large multi-node settings (up to 32 nodes), which is increasingly relevant. We study and identify scaling bottlenecks for both TP and PP. For decode-bound workloads, we further characterize PP's limited ability to reduce computation time and attribute TP's communication overhead to sub-optimal NCCL all-reduce performance in the small message regime.

Collective Communication Optimization. Recent work explores collective communication optimizations to improve distributed inference. A popular approach is to overlap communication with computation. ISO [47] achieves overlap in prefill, but not in decode, where our approach is most beneficial. Ladder-residual [50] achieves overlap through model-architecture changes, which we avoid. StragglAR [8] proposes a new all-reduce algorithm to reduce stragglers in bandwidth-bound regimes, whereas NVRAR focuses on latency-bound regimes. Other low-latency all-reduce optimizations, such as those in vLLM [18] and TensorRT-LLM [29], target NVLink-connected domains. NVRAR targets scale-out networks and is complementary to these approaches — its intra-node phases can be replaced with NVLink-optimized variants. Other recent works [2, 51, 54] explore NVSHMEM-based collective optimizations for all-to-all communication in EP deployments, which are orthogonal to NVRAR's optimizations for TP all-reduce. Finally, several works [17, 41] employ hierarchical communication patterns for large bandwidth-bound messages typical of training workloads. In contrast, we focus on inference workloads and target the latency-sensitive small message regime.

7 Conclusion

In this work, we conduct a detailed performance study of model-parallelism schemes for multi-node LLM inference workloads. We compare the performance of Tensor Parallelism (TP) and Hybrid Parallelism (TP+PP) across batched inference workloads and identify the scaling bottlenecks for each strategy. Focusing on workloads that favor TP, we observe severe communication bottlenecks arising from sub-optimal NCCL all-reduce performance for small message sizes. Motivated by this, we develop NVRAR, a hierarchical all-reduce implementation built using NVSHMEM. We make several key optimizations in NVRAR to reduce latencies for small-message

all-reduce operations. Compared to NCCL, NVRAR achieves 1.06×–1.92× lower latency on Slingshot and 1.14×–3.57× on InfiniBand for 256 KB–2 MB messages. Integrated into YALIS and vLLM, NVRAR significantly improves multi-node TP inference for large dense and MoE models on realistic workloads.

Acknowledgments

This work was performed in part under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-2013350).

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility, operated under Contract No. DE-AC02-05CH11231 using NERSC award DDR-ERCAP0034262 and ALCC-ERCAP0034775. This research is supported by the National Artificial Intelligence Research Resource (NAIRR) Pilot and used the Delta advanced computing and data resource which is supported by the NSF (award NSF-OAC 2005572) and the State of Illinois, and the Vista supercomputing resource at the Texas Advanced Computing Center (TACC) at The University of Texas at Austin. The authors acknowledge the University of Maryland supercomputing resources made available for conducting the research reported in this paper. This work was supported by a grant from the Swiss National Supercomputing Centre (CSCS) under project ID lp98 on Alps.

References

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

[2] Osayamen Jonathan Aimuyo, Byungsoo Oh, and Rachee Singh. 2025. FlashDMoE: Fast Distributed MoE in a Single Kernel. https://arxiv.org/abs/2506.04667

[3] Eduardo Alvarez. 2025. Analyzing the Impact of Tensor Parallelism Configurations on LLM Inference Performance. AMD ROCm Blogs. (March 2025). https://rocm.blogs.amd.com/artificial-intelligence/tensor-parallelism/README.html

[4] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. 2022. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[5] Abhinav Bhatele, Rakrish Dhakal, Alexander Movsesyan, Aditya K. Ranjan, and Onur Cankur. 2023. Pipit: Scripting the analysis of parallel execution traces. arXiv:2306.11177 [cs.DC]

[6] Alexander Bick, Adam Blandin, and David J Deming. 2024. *The rapid adoption of generative AI.* Technical Report. National Bureau of Economic Research.

[7] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. 2010. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model* (New York, New York, USA) *(PGAS '10)*. Association for Computing Machinery, New York, NY, USA, Article 2, 3 pages. doi:10.1145/2020373.2020375

[8] Arjun Devraj, Eric Ding, Abhishek Vijaya Kumar, Robert Kleinberg, and Rachee Singh. 2025. Accelerating AllReduce with a Persistent Straggler. https://arxiv.org/abs/2505.23523

[9] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[10] 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

[11] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).

[12] Roger W. Hockney. 1994. The communication challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Comput.* 20, 3 (March 1994), 389–398. doi:10.1016/S0167-8191(06)80021-9

[13] 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.

[14] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. arXiv:1811.06965 [cs.CV]

[15] International Energy Agency. 2025. Energy and AI. https://www.iea.org/reports/energy-and-ai Licence: CC BY 4.0.

[16] Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. 2024. Openai o1 system card. *arXiv preprint arXiv:2412.16720* (2024).

[17] Youhe Jiang, Huaxi Gu, Yunfeng Lu, and Xiaoshan Yu. 2020. 2D-HRA: Two-Dimensional Hierarchical Ring-Based All-Reduce Algorithm in Large-Scale Distributed Machine Learning. *IEEE Access* 8 (2020), 183488–183494. doi:10.1109/ACCESS.2020.3028367

[18] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles.*

[19] Qingyuan Li, Bo Zhang, Liang Ye, Yifan Zhang, Wei Wu, Yerui Sun, Lin Ma, and Yuchen Xie. 2024. Flash Communication: Reducing Tensor Parallelization Bottleneck for Fast Large Language Model Inference. arXiv:2412.04964 [cs.AI] https://arxiv.org/abs/2412.04964

[20] Lightning AI. 2023. LitGPT. https://github.com/Lightning-AI/litgpt.

[21] Nestor Maslej, Loredana Fattorini, Raymond Perrault, Yolanda Gil, Vanessa Parli, Njenga Kariuki, Emily Capstick, Anka Reuel, Erik Brynjolfsson, John Etchemendy, Katrina Ligett, Terah Lyons, James Manyika, Juan Carlos Niebles, Yoav Shoham, Russell Wald, Tobi Walsh, Armin Hamrah, Lapo Santarlasci, Julia Betts Lotufo, Alexandra Rome, Andrew Shi, and Sukrut Oak. 2025. Artificial Intelligence Index Report 2025. arXiv:2504.07139 [cs.AI] https://arxiv.org/abs/2504.07139

[22] Meta. 2023. Torch Compile. https://docs.pytorch.org/tutorials/intermediate/torch_compile_tutorial.html.

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

[24] NVIDIA. [n. d.]. NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems.

[25] NVIDIA. 2017. NCCL Tests. https://github.com/NVIDIA/nccl-tests.

[26] NVIDIA. 2019. CUDA Graphs. https://developer.nvidia.com/blog/cuda-graphs/.

[27] NVIDIA. 2019. Massively Scale Your Deep Learning Training with NCCL 2.4. https://developer.nvidia.com/blog/massively-scale-deep-learning-training-nccl-2-4/.

[28] NVIDIA. 2020. NVSHMEM. https://developer.nvidia.com/nvshmem.

[29] NVIDIA. 2023. TensorRT-LLM. https://github.com/NVIDIA/TensorRT-LLM

[30] 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

[31] Ray Project Contributors. 2025. Ray GitHub Issue #58426. https://github.com/ray-project/ray/issues/58426.

[32] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300* (2024).

[33] 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] https://doi.org/10.48550/arXiv.1909.08053

[34] Siddharth Singh and Abhinav Bhatele. 2022. AxoNN: An asynchronous, message-driven parallel framework for extreme-scale deep learning. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium (IPDPS '22)*. IEEE Computer Society. https://doi.org/10.1109/IPDPS53621.2022.00065

[35] Siddharth Singh, Prajwal Singhania, Aditya Ranjan, John Kirchenbauer, Jonas Geiping, Yuxin Wen, Neel Jain, Abhimanyu Hans, Manli Shu, Aditya Tomar, Tom Goldstein, and Abhinav Bhatele. 2024. Democratizing AI: Open-source Scalable LLM Training on GPU-based Supercomputers. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '24)*. https://doi.org/10.1109/SC41406.2024.00010

[36] Siddharth Singh, Prajwal Singhania, Aditya K. Ranjan, Zack Sating, and Abhinav Bhatele. 2024. A 4D Hybrid Algorithm to Scale Parallel Training to Thousands of GPUs. arXiv:2305.13525 [cs.LG]

[37] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2024. Scaling LLM Test-Time Compute Optimally can be More Effective than Scaling Model Parameters. arXiv:2408.03314 [cs.LG] https://arxiv.org/abs/2408.03314

[38] Benjamin Spector, Jordan Juravsky, Stuart Sul, Dylan Lim, Owen Dugan, Simran Arora, and Chris Ré. 2025. We Bought the Whole GPU, So We're Damn Well Going to Use the Whole GPU. https://hazyresearch.stanford.edu/blog/2025-09-28-tp-llama-main Hazy Research Blog.

[39] Qidong Su, Wei Zhao, Xin Li, Muralidhar Andoorveedu, Chenhao Jiang, Zhanda Zhu, Kevin Song, Christina Giannoula, and Gennady Pekhimenko. 2025. Seesaw: High-throughput llm inference via model re-sharding. *arXiv preprint arXiv:2503.06433* (2025).

[40] Rajeev Thakur and William D. Gropp. 2003. Improving the Performance of Collective Operations in MPICH. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Jack Dongarra, Domenico Laforenza, and Salvatore Orlando (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 257–267.

[41] Yuichiro Ueno and Rio Yokota. 2019. Exhaustive Study of Hierarchical AllReduce Patterns for Large Messages Between GPUs. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 430–439. doi:10.1109/CCGRID.2019.00057

[42] Ohio State University. [n. d.]. OSU Micro-Benchmarks 5.8. http://mvapich.cse.ohio-state.edu/benchmarks/.

[43] vLLM. 2023. vLLM Benchmark CLI. https://docs.vllm.ai/en/stable/benchmarking/cli/.

[44] vLLM Team. 2024. Announcing Llama 3.1 Support in vLLM. *vLLM Blog*. https://blog.vllm.ai/2024/07/23/llama31.html. https://blog.vllm.ai/2024/07/23/llama31.html Accessed: 2025-10-30.

[45] Yuxin Wang, Yuhan Chen, Zeyu Li, Xueze Kang, Yuchu Fang, Yeju Zhou, Yang Zheng, Zhenheng Tang, Xin He, Rui Guo, et al. 2025. Burstgpt: A real-world workload dataset to optimize llm serving systems. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*. 5831–5841.

[46] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[47] Bin Xiao and Lei Su. 2024. ISO: Overlap of Computation and Communication within Sequence for LLM Inference. https://arxiv.org/abs/2409.11155

[48] Lang Xu, Kaushik Kandadi Suresh, Quentin Anthony, Nawras Alnaasan, and Dhabaleswar K Panda. 2025. Characterizing Communication Patterns in Distributed Large Language Model Inference. *arXiv preprint arXiv:2507.14392* (2025).

[49] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 521–538. https://www.usenix.org/conference/osdi22/presentation/yu

[50] Muru Zhang, Mayank Mishra, Zhongzhu Zhou, William Brandon, Jue Wang, Yoon Kim, Jonathan Ragan-Kelley, Shuaiwen Leon Song, Ben Athiwaratkun, and Tri Dao. 2025. Ladder-residual: parallelism-aware architecture for accelerating large model inference with communication overlapping. *arXiv preprint arXiv:2501.06589* (2025).

[51] 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).

[52] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2024. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems* 37 (2024), 62557–62583.

[53] Kan Zhu, Yufei Gao, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, et al. 2025. {NanoFlow}: Towards optimal large language model serving throughput. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. 749–765.

[54] Ruidong Zhu, Ziheng Jiang, Chao Jin, Peng Wu, Cesar A. Stuardo, Dongyang Wang, Xinlei Zhang, Huaping Zhou, Haoran Wei, Yang Cheng, Jianzhe Xiao, Xinyi Zhang, Lingjun Liu, Haibin Lin, Li-Wen Chang, Jianxi Ye, Xiao Yu, Xin Liu, and Xin Jin. 2025. MegaScale-Infer: Serving Mixture-of-Experts at Scale with Disaggregated Expert Parallelism. https://arxiv.org/abs/2504.02263

A Extended Performance Study Results

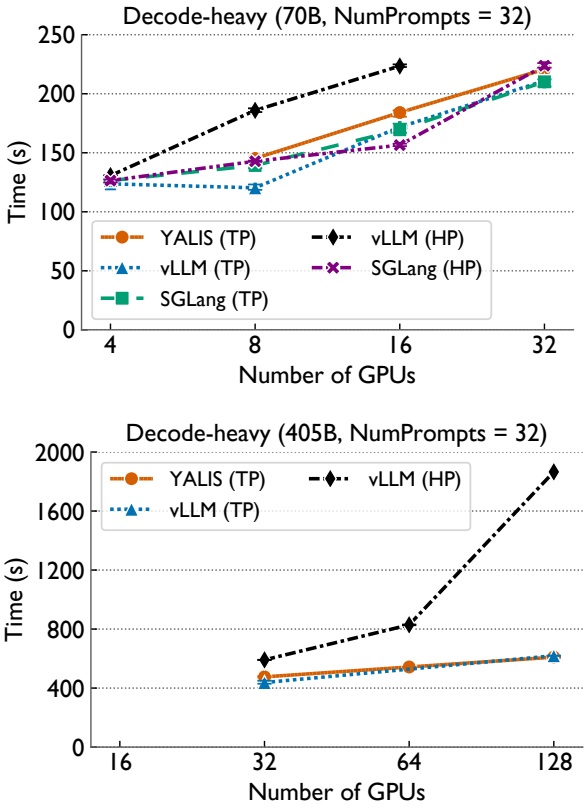

Figure 11: Strong scaling performance of different inference engines on Perlmutter for the Llama 3.1 70B (top) and 405B (bottom) models, for the decode-heavy workload with NumPrompts = 32. The Y-axis denotes the time to completion for a batch of prompts in seconds and the X-axis denotes the number of GPUs.

Figure 11 presents the strong scaling performance of different inference engines on Perlmutter for the Llama 3.1 70B and 405B models, for the decode-heavy workload with NumPrompts = 32. We observe that vLLM V0 (HP) (black line) scales poorly for both the 70B and 405B models. Both TP and HP scale poorly, with total time increasing as the number of GPUs increases for the 70B model (top). Unlike vLLM V0 (HP), SGLang (HP) has runtimes closer to the TP configurations for the 70B model, but it still scales poorly. This closely matches our observation in the main text.

B Reconciling Microbenchmark and End-to-End Performance

In this section, we resolve the discrepancy observed in the speedup of NVRAR over NCCL all-reduce in the standalone benchmark compared to the end-to-end workload. For the standalone benchmark, we launch communication kernels back-to-back, in a single CUDA Graph, without any computation work in between. This is not representative of the real-world use cases where collective communication operations are interleaved with computation. Figure 12 illustrates an example YALIS trace where the all-reduce operation is followed by several matmul and compute kernels. We account for this difference by running synthetic matrix multiplications between the all-reduce calls in our standalone benchmark.

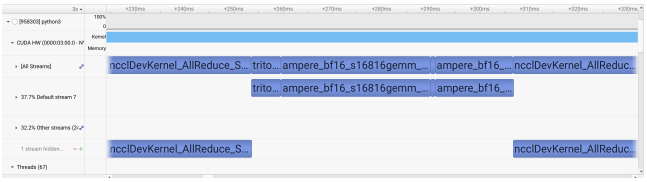

Figure 12: NSYS profile snapshots of the NVRAR kernel in the microbenchmark setup (left) and YALIS (right).

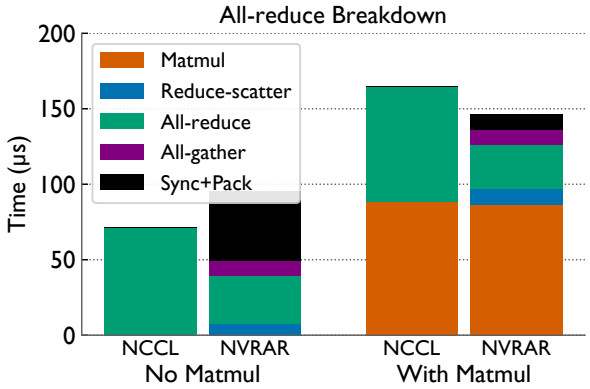

Figure 13: All-reduce time breakdown of NVRAR and NCCL for a 128 KB message on Perlmutter (16 GPUs), with and without interleaved matmul computation.

Figure 13 breaks down the 128 KB all-reduce microbenchmark on Perlmutter (16 GPUs) for NVRAR and NCCL, without and with an interleaved representative matmul. Without matmul, NVRAR spends a large fraction of time in synchronization. With interleaved matmuls, the synchronization costs are hidden by the matmul computation due to the deferred peer-wise synchronization strategy of NVRAR.

C Extended NVRAR Evaluation

This section presents an extended evaluation of NVRAR's performance, both as a standalone collective against NCCL in microbenchmarks and as part of tensor-parallel inference workloads in YALIS and vLLM.

C.1 Impact of Chunk Size and Block Size on NVRAR Performance

We demonstrate that the hyperparameters — block size (B_s) and chunk size (C_s) — have a significant impact on the performance of NVRAR. To tune these hyperparameters, we run NVRAR with different values of B_s and C_s for different message sizes and node counts. Table 5 reports the performance of NVRAR with four different hyperparameter configurations for an all-reduce message size of 1024 KB on 16 GPUs. We observe that the performance is

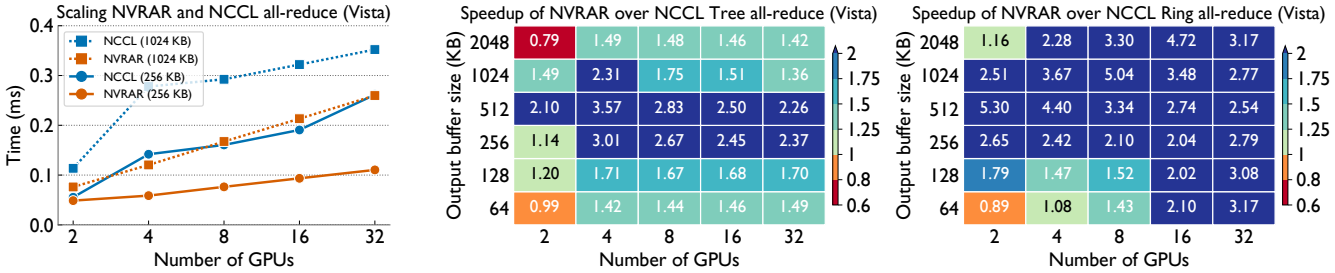

Figure 14: Performance comparison of NVRAR and NCCL all-reduce on Vista. (Left) Scaling of NVRAR and NCCL all-reduce for 256 KB and 1024 KB messages across GPU counts. (Middle, Right) Speedup of NVRAR over NCCL all-reduce with the NCCL algorithm fixed to Tree and Ring, respectively, across message sizes and GPU counts.

more sensitive to changing the chunk size (C_s) than the block size (B_s). This validates our design choice of keeping these hyperparameters tunable. We also observe that chunking large messages within each block improves performance, with smaller chunk sizes outperforming some larger chunk sizes. We leave heuristic-based hyperparameter tuning to future work.

Table 5: Different NVRAR hyperparameter configurations for an all-reduce message size of 1024 KB on 16 GPUs.

Block Size (B_s)	Chunk Size (C_s)	Time (ms)
32	32768	0.1522 ms
32	4096	0.2271 ms
8	16384	0.1891 ms
8	131072	0.1655 ms

C.2 Phase-Wise NVRAR Breakdown

Figure 13 also presents the breakdown of the time spent in each phase of NVRAR. For the 128 KB message size, we observe that most of the time is spent in the inter-node recursive-doubling phase. This is expected as the communication happens on the slower Slingshot/InfiniBand network, as compared to the intra-node phases that happen on the faster NVLink network.

C.3 Comparing NVRAR and NCCL All-reduce

In this section, we present additional microbenchmark results on Vista and studies that isolate the impact of NCCL's algorithm selection and library version on the observed speedups.

C.3.1 Scaling Results on Vista. Figure 14 (left) reports the all-reduce microbenchmark performance for 256 KB and 1024 KB messages on Vista, complementing the Perlmutter results in the main text (Figure 6). NVRAR scales similarly on both platforms and outperforms NCCL across all GPU counts.

C.3.2 NCCL Algorithm Selection. To ensure that NVRAR's performance gains are not an artifact of NCCL's automatic algorithm selection, we force NCCL to use the Tree and Ring algorithms individually and compare against NVRAR. Figure 14 (middle, right) reports the results on Vista. NVRAR remains faster than NCCL across the GPU range and message sizes for both algorithms, confirming that the speedups reported in the main text are not an artifact of NCCL's algorithm selection.

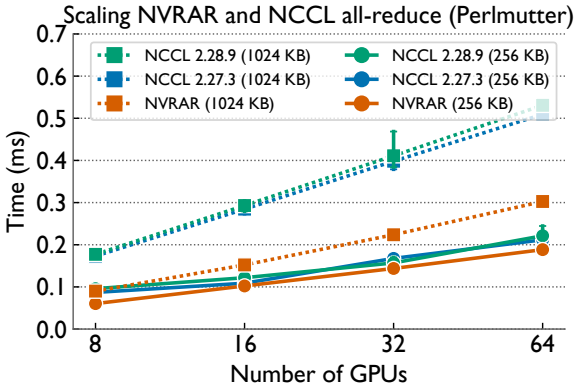

Figure 15: All-reduce scaling of NVRAR against two NCCL versions (2.27.3 and 2.28.9) on Perlmutter for 256 KB and 1024 KB messages.

C.3.3 Comparison Across NCCL Versions. The microbenchmark results reported in the main text use NCCL 2.27.3. Figure 15 compares its all-reduce performance against NCCL 2.28.9 (shipped with the latest PyTorch 2.11) on Perlmutter, alongside NVRAR. The two NCCL versions track each other closely across all GPU counts and both message sizes, and NVRAR retains its speedup over both. This confirms that the speedups reported in the main text are not limited to the specific NCCL version we evaluated against. While newer NCCL versions have several performance improvements, most of the all-reduce optimizations are targeted towards homogeneous NVLink-connected systems (intra-node or multi-node NVLink systems) and are orthogonal to NVRAR's optimizations for heterogeneous networks.

C.4 Multi-node Inference Evaluation

In this section, we present the extended performance evaluation when using NVRAR with YALIS and vLLM in multi-node inference settings to complement the results in Section 5.

C.4.1 Batched Inference Performance on Vista. Figure 16 reports the relative speedup of YALIS (TP) using NVRAR all-reduce over YALIS (TP) using NCCL all-reduce for the decode-heavy workload on Vista. We observe that for the 70B model, NVRAR achieves a speedup of 1.92× for NumPrompts=32 on 16 GPUs. This is consistent with our observation on Perlmutter in the main text, validating that NVRAR provides performance benefits on both Slingshot and InfiniBand networks.

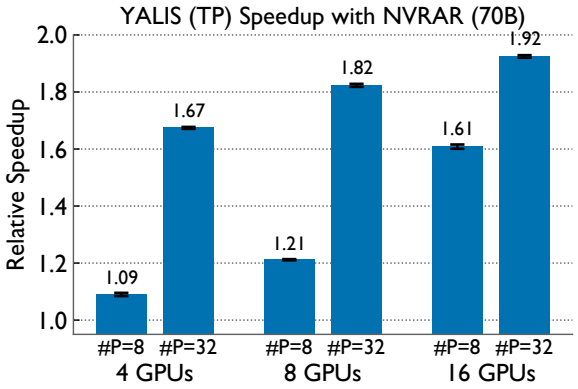

Figure 16: Relative speedup of YALIS (TP) using NVRAR all-reduce over YALIS (TP) using NCCL all-reduce for the decode-heavy workload on Vista.

C.4.2 BurstGPT Trace Details. Figure 17 presents input sequence length and output sequence length distributions for the BurstGPT trace used in the trace-based performance evaluation (Figure 9). Table 6 provides the details of the vLLM benchmark configuration.

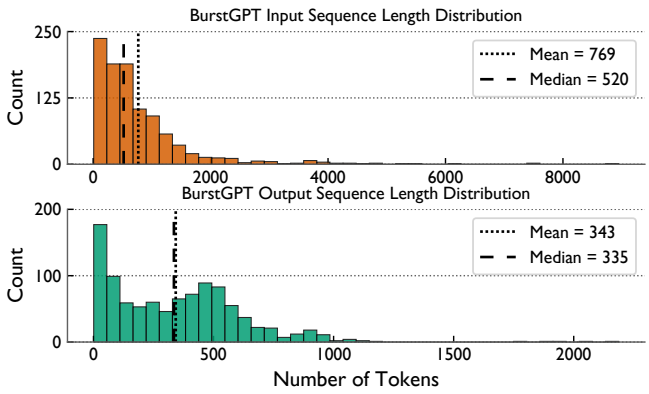

Figure 17: Input sequence length and output sequence length distribution for the BurstGPT trace (1,000 prompts).

C.4.3 Evaluation on Decode-heavy Trace. In addition to BurstGPT, we evaluate on a randomly generated decode-heavy trace with mean input and output sequence lengths of 1024 and 4096, respectively. We use the same vLLM benchmark settings as in Table 6.

Figure 18 reports the output throughput of NVRAR-based TP, NCCL-based TP, and HP for the decode-heavy trace. We observe

Table 6: vLLM benchmark settings for trace-based performance evaluation.

Setting	Value
Concurrency	32, 256
Number of Prompts	1,000
Configured Request Trace	10 requests/second
Burstiness	2.0 (Gamma distribution)

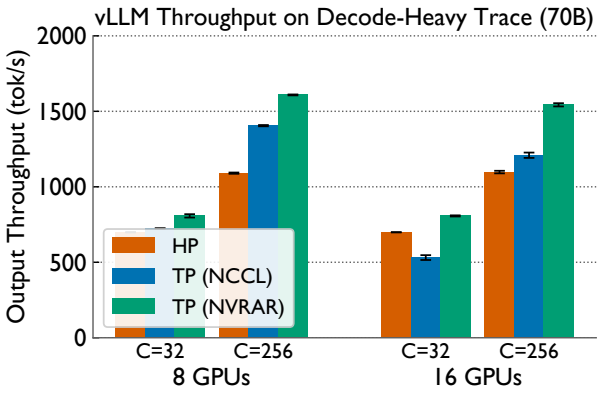

Figure 18: Output throughput on a randomly generated decode-heavy trace when serving the 70B model on Perlmutter with vLLM. We compare NCCL-based TP, NVRAR-based TP, and HP deployments under two maximum request concurrency settings (C=32 and C=256).

that in this case, NVRAR-based TP outperforms NCCL-based TP by 1.11×–1.52× and HP by 1.16×–1.48× across these settings, and outperforms the best NCCL-based deployment by 1.11×–1.28×. These are higher than the speedups observed on the BurstGPT trace (Figure 9). This is expected, as NVRAR is more beneficial in the decode phase, with small to medium message sizes. This is consistent with our batched inference performance evaluations.