Faculty of Engineering Sciences

Heidelberg University

Master Thesis

in Computer Engineering

submitted by

Leandro Borzyk

born in Speyer, Germany

20/02/2026

UNDERSTANDING LLM COMMUNICATION

This Master thesis has been carried out by Leandro Borzyk
at the
Hardware and Artificial Intelligence (HAWAII) Lab
at the Institute of Computer Engineering
Supervisor: Prof. Dr. Holger Fröning
Second Examiner: TBD

ABSTRACT

The training of modern Large Language Models (LLMs) requires distributed computing across Graphics Processing Unit (GPU) clusters, where network communication efficiency critically impacts performance and cost. Existing profiling tools provide either high-level metrics or low-level timing data, but lack the operation-level granularity needed to understand communication patterns during training runs. This thesis presents the **NCCL Trace Profiler**, a novel methodology for fine-grained analysis of network communication during distributed Large Language Model (LLM) training. The core contribution is a sequence alignment approach that correlates NVIDIA Collective Communications Library (NCCL) debug logs with NVIDIA Nsight Systems (Nsys) kernel traces, fusing semantic metadata with nanosecond-accurate timing without code instrumentation. The resulting Python tool works with standard profiling outputs, implements automated topology detection, and produces enriched traces for interactive analysis of communication behavior. Using the profiler, this thesis conducts a systematic characterization of communication patterns across Data Parallelism (DP), Tensor Parallelism (TP), Pipeline Parallelism (PP), and Expert Parallelism (EP) in NVIDIA's Megatron-LM framework. The analysis documents operation type distributions, message size characteristics, and per-rank communication volumes, providing the first detailed, per-operation view of how parallelization strategies manifest as network traffic signatures. Furthermore, the work validates theoretical communication volume models against observed measurements, revealing that while the Data Parallelism (DP) model achieves excellent accuracy, Tensor Parallelism (TP) and Expert Parallelism (EP) models exhibit systematic underestimation due to fine-grained synchronization and routing overhead not captured in idealized models.

ZUSAMMENFASSUNG

Das Training moderner Large Language Models (LLMs) erfordert verteiltes Rechnen über Graphics Processing Unit (GPU)-Cluster hinweg, wobei die Effizienz der Netzwerkkommunikation einen entscheidenden Einfluss auf Leistung und Kosten hat. Bestehende Profiling-Tools liefern entweder hochrangige Metriken oder niedrigrangige Zeitdaten, verfügen jedoch nicht über die erforderliche Granularität auf Operationsebene, um Kommunikationsmuster während des Trainings zu verstehen. Diese Arbeit stellt den **NCCL Trace Profiler** vor, eine neuartige Methodik zur detaillierten Analyse der Netzwerkkommunikation während des verteilten LLM-Trainings. Der zentrale Beitrag ist ein Sequenzabgleichansatz, der NVIDIA Collective Communications Library (NCCL)-Debug-Protokolle mit NVIDIA Nsight Systems (Nsys)-Kernel-Traces korreliert und semantische Metadaten mit nanosekundengenauen Zeitangaben ohne Code-Instrumentierung zusammenführt. Das daraus resultierende Python-Tool arbeitet mit Standard-Profiling-Ausgaben, implementiert eine automatisierte Topologieerkennung und erzeugt angereicherte Traces für die interaktive Analyse des Kommunikationsverhaltens. Mit Hilfe des Profilers führt diese Arbeit eine systematische Charakterisierung der Kommunikationsmuster über Data Parallelism (DP), Tensor Parallelism (TP), Pipeline Parallelism (PP) und Expert Parallelism (EP) im Megatron-LM-Framework von NVIDIA durch. Die Analyse dokumentiert die Verteilung der Operationstypen, die Merkmale der Nachrichtengröße und das Kommunikationsvolumen pro Rang und liefert damit erstmals einen detaillierten Überblick darüber, wie sich Parallelisierungsstrategien in Form von Netzwerkverkehrssignaturen manifestieren. Darüber hinaus validiert die Arbeit theoretische Kommunikationsvolumenmodelle anhand beobachteter Messungen und zeigt, dass das DP-Modell zwar eine ausgezeichnete Genauigkeit erzielt, die Modelle für TP und EP jedoch aufgrund der feinkörnigen Synchronisation und des Routing-Overheads, die in idealisierten Modellen nicht erfasst werden, eine systematische Unterschätzung aufweisen.

ACKNOWLEDGMENTS

The author acknowledges support by the state of Baden-Württemberg through bwHPC and the German Research Foundation (DFG) through grant INST 35/1597-1 FUGG.

CONTENTS

1 Introduction 1
- 1.1 Problem Statement 2
- 1.2 Research Objectives and Approach 2
- 1.3 Contributions 3

2 Background 5
- 2.1 From Transformers to LLMs 5
- 2.2 Hardware Infrastructure for LLM Training 7
  - 2.2.1 GPU Accelerators 7
  - 2.2.2 Node Configuration 7
  - 2.2.3 Interconnect Technologies 8
- 2.3 NCCL: GPU Collective Communication Library 9
  - 2.3.1 Design Principles and Architecture 9
  - 2.3.2 Collective Operations 9
  - 2.3.3 Protocol Variants and Topology Awareness 11
  - 2.3.4 Integration 12
- 2.4 Distributed Training Strategies and Communication Patterns 12
  - 2.4.1 Data Parallelism 13
  - 2.4.2 Tensor Parallelism 15
  - 2.4.3 Pipeline Parallelism 16
  - 2.4.4 Sequence and Context Parallelism 19
  - 2.4.5 Expert Parallelism 19
  - 2.4.6 Multidimensional Parallelism 21
- 2.5 The Needleman-Wunsch Algorithm 23
  - 2.5.1 Problem Formulation 24
  - 2.5.2 Dynamic Programming Solution 25
  - 2.5.3 Application to Trace Alignment 26

3 State of the Art and Related Work 27
- 3.1 Literature Review Methodology 27
- 3.2 State of the Art in Distributed Training Profiling 28
  - 3.2.1 Characterization and Performance Modeling Studies 29
  - 3.2.2 Communication Analysis in Production Systems 29
  - 3.2.3 Framework-Specific Profiling and Debugging 30
  - 3.2.4 Communication Overhead Quantification 30
- 3.3 Existing Profiling Tools and Technologies 31
  - 3.3.1 Framework-Level Profilers 31
  - 3.3.2 Hardware-Level Profilers 31
  - 3.3.3 Communication Library Profiling 32
- 3.4 The Research Gap 33

4 NCCL Trace Profiler 37
4.1 The Trace Correlation Challenge 37
4.1.1 Why Direct Correlation Fails 37
4.1.2 Sequence Alignment as the Solution Approach 40
4.2 Profiler Architecture 41
4.2.1 Data Ingestion and Parsing 42
4.2.2 Sequence Alignment via Needleman-Wunsch 46
4.2.3 Communicator Inference and Global Rank Assignment 48
4.2.4 Clock Synchronization 49
4.2.5 Bandwidth Calculation and Theoretical Limits 50
4.2.6 Chrome Trace Export and Visualization 52
4.3 Validation 53
4.3.1 Alignment Algorithm Validation 53
4.3.2 Integration Testing: Comparative Scenarios 55
4.3.3 Component-Level Tests 57
5 LLM Communication Analysis 59
5.1 Experimental Infrastructure: Node Architectures 59
5.1.1 PCIe-Only Architecture: NVIDIA A40 59
5.1.2 Full NVLink Mesh: NVIDIA A100-SXM4 (40Gigabyte (GB)) 60
5.1.3 Architectural Implications 60
5.1.4 Hardware Characterization and Baselines 61
5.2 Experimental Methodology 63
5.2.1 Model Architectures 63
5.2.2 Training Configuration and Hyperparameters 64
5.2.3 Distributed Training Framework 65
5.2.4 Execution 65
5.3 Communication Pattern Analysis 65
5.3.1 Data Parallelism (DP=4) 66
5.3.2 Pipeline Parallelism (PP=4) 70
5.3.3 Tensor Parallelism (TP=4) 75
5.3.4 Expert Parallelism (EP=4) 79
5.3.5 Theoretical Validation of Communication Volume 82
6 Discussion and Outlook 91
6.1 Current Limitations 92
6.2 Future Directions 94
6.3 Conclusion 94
A Appendix 101
A.1 Detailed Needleman Wunsch Example 101
A.1.1 Step-by-Step Matrix Construction 101
A.1.2 Filling the Matrix 101
A.1.3 Traceback and Result 102
A.2 Environment Setup Script 103

A.3 Training Script 105

Bibliography 115

1 INTRODUCTION

The advancement of Artificial Intelligence (AI) has become synonymous with the rapid evolution of deep learning models. In recent years, the scale of these models, particularly in the domain of Natural Language Processing, has expanded exponentially, with leading Large Language Models (LLMs) now encompassing hundreds of billions or even trillions of parameters. The computational and memory requirements for training such architectures far exceed the capacity of any single processing unit. Consequently, distributed training across large-scale clusters of Graphics Processing Units (GPUs) has transitioned from a niche high-performance computing technique to a fundamental prerequisite for state-of-the-art AI research and development. In this distributed paradigm, the network fabric connecting the Graphics Processing Units (GPUs) becomes a critical infrastructure component, where efficiency often determines training scalability and time-to-solution.

To facilitate the training of these massive models, the research community has developed sophisticated parallelism frameworks, with NVIDIA's Megatron-LM and Microsoft's DeepSpeed serving as prominent examples. These frameworks employ a combination of strategies to distribute both computation and memory across devices. The most established approach, Data Parallelism (DP), replicates the entire model on each GPU while partitioning the global data batch across devices, a well-understood technique that scales naturally as more GPUs are added. However, when individual models grow too large to fit within a single GPU's memory, more sophisticated decomposition methods become necessary. Tensor Parallelism (TP) addresses this challenge by partitioning individual model layers horizontally across multiple GPUs, enabling the training of layers that would otherwise exceed device memory limits. Complementing this horizontal partitioning, Pipeline Parallelism (PP) takes a vertical approach by dividing the model's layers into sequential stages, forming a computational pipeline where different GPUs process different stages of the forward and backward passes concurrently.

The intricate communication patterns demanded by these parallelism strategies are orchestrated by the NVIDIA Collective Communications Library (NCCL), which provides highly optimized implementations of collective primitives such as `AllReduce` and `AllGather`. These operations are fundamental for synchronizing parameters and

gradients across devices, and their efficiency directly impacts overall training throughput.

1.1 PROBLEM STATEMENT

Despite established performance benchmarks demonstrating the scalability of these frameworks, much of the prior work treats the underlying network communication as a "black box". A review of existing literature reveals that research in large-scale model training has predominantly focused on high-level performance metrics. Numerous studies have benchmarked the end-to-end training throughput and scalability of frameworks like Megatron-LM and DeepSpeed. While these works successfully demonstrate the efficacy of new parallelism algorithms by measuring their impact on global training time, they rarely scrutinize the low-level network traffic patterns that drive these results.

This gap between high-level metrics and low-level behavior creates significant challenges for practitioners attempting to optimize distributed training workloads. Existing profiling tools provide incomplete information: framework-level profilers capture high-level operator semantics, while hardware profilers like NVIDIA Nsight Systems (Nsys) provide nanosecond-accurate kernel timing but no information about which communicator or message size is associated with each operation.

As training runs for frontier models can cost millions of dollars in compute time, even small improvements in communication efficiency translate to significant cost savings and faster iteration cycles for AI research. The ability to profile and analyze communication patterns at operation-level granularity is essential for identifying bottlenecks, validating performance models, as well as optimizing hardware setup and parallelization strategies.

1.2 RESEARCH OBJECTIVES AND APPROACH

This thesis addresses the profiling gap by developing a novel methodology for fine-grained analysis of NVIDIA Collective Communications Library (NCCL) communication patterns during distributed LLM training.

The core insight is that the information needed for detailed communication analysis exists, but is split across two incompatible data sources: NCCL debug logs contain semantic metadata (operation types, message sizes, communicators) but lack precise timing, while NVIDIA Nsight Systems (Nsys) kernel traces provide nanosecond-accurate timing but no semantic context. By treating the correlation of these

two data sources as a sequence alignment problem, this work develops an automated methodology that fuses complementary information without requiring code instrumentation or custom NVTX parsing.

The resulting **NCCL Trace Profiler** is a Python-based tool that works out-of-the-box with standard profiling outputs, producing enriched traces that combine semantic metadata with precise timing information. This enables practitioners to move beyond aggregate performance metrics and understand the precise sequence, timing, and structure of network operations during distributed training.

Using this profiler, the thesis conducts a systematic characterization of communication patterns across different parallelism strategies in Megatron-LM, documenting the network traffic signatures, communication volumes, and structural characteristics of DP, TP, Pipeline Parallelism (PP), and Expert Parallelism (EP) configurations, and validating theoretical communication models against observed behavior.

1.3 CONTRIBUTIONS

The primary contributions of this thesis are:

1. **NCCL Trace Profiler Design and Implementation:** A Python-based profiling tool that correlates NCCL debug logs with Nsys kernel traces using a sequence alignment algorithm adapted from bioinformatics. The profiler operates entirely post hoc, requiring no modifications to application or framework source code, and consumes standard profiling outputs. It produces enriched, temporally aligned traces in the Chrome Trace Event Format to support interactive inspection of distributed training communication behavior.

2. **Systematic Characterization of Parallelism Communication Patterns:** Using the developed profiler, this work systematically measures and characterizes the network traffic signatures produced by DP, TP, PP, and EP in Megatron-LM training workloads. The analysis documents operation type distributions, message size characteristics, temporal patterns, and per-rank communication volumes across different parallelism strategies and hardware configurations. Furthermore, the work validates theoretical communication volume formulas against observed measurements, quantifying the gap between idealized models and implementation reality. This empirical characterization provides the first fine-grained, per-operation view of how parallelization decisions manifest as actual network behavior during LLM training.

3. **Topology Detection and Theoretical Bandwidth Calculation:** The profiler implements automated hardware topology detection from NCCL debug logs, extracting interconnect bandwidths and

link types. This topology information, combined with precise timing data, enables future per-operation bandwidth analysis.

The thesis is structured as follows: Chapter 2 provides the necessary background on distributed deep learning, the NCCL library, parallelism strategies, and the Needleman Wunsch alignment algorithm. Chapter 3 presents related work in distributed training profilers and trace analysis tools. Chapter 4 details the design and implementation of the NCCL Trace Profiler, including the sequence alignment methodology, topology detection, and output visualization (Contributions 1 and 3). Chapter 5 describes the experimental methodology and presents the results of the traffic characterization study (Contribution 2). Finally, Chapter 6 concludes the thesis, summarizing key findings and discussing potential avenues for future work.

2 BACKGROUND

This chapter provides the necessary background to understand the computational and communication challenges in training large language models. It begins by introducing modern LLMs and their unprecedented computational requirements, then describes the GPU cluster hardware infrastructure that enables distributed training. Next, it examines NCCL, the communication library that orchestrates GPU-to-GPU data transfer, followed by the distributed training strategies that partition computation and data across the cluster. Finally, it introduces the Needleman-Wunsch sequence alignment algorithm, which provides the methodological foundation for the trace correlation approach developed in this thesis.

2.1 FROM TRANSFORMERS TO LLMS

The Transformer architecture introduced by Vaswani et al. [82] was originally designed for machine translation using an encoder-decoder structure. Subsequent work demonstrated that the decoder-only variant, when trained at scale on vast text corpora, could serve as a powerful general-purpose language model, leading to the development of the Generative Pre-trained Transformer (GPT) series by OpenAI [3, 62, 63].

GPT models adopt a decoder-only architecture with unidirectional self-attention, where each token attends only to previous tokens in the sequence. This enables autoregressive generation with a simple next-token prediction objective:

$$\mathcal{L} = -\sum_{i=1}^{N} \log P(x_i \mid x_1, \ldots, x_{i-1}; \theta) \tag{2.1}$$

where x_i denotes the i-th token in the input sequence, $\{x_1, \ldots, x_N\}$ is the tokenized training sequence, θ represents the model parameters, and N is the sequence length.

The key insight was that scaling these models in both parameter count and training data, led to emergent capabilities not present in smaller models [3, 84]. GPT-3 [3], with 175 billion parameters, demonstrated impressive few-shot learning abilities, performing novel tasks with minimal examples provided in the prompt.

Contemporary LLMs have continued this scaling trend, reaching hundreds of billions or even trillions of parameters. Notable examples include Google's Gemini [6, 75], Meta's LLaMA [78, 79], OpenAI's GPT [56, 57], and DeepSeek's models [26, 27]. These models are trained on diverse datasets comprising trillions of tokens from web pages, books, scientific articles, and code repositories.

As model and dataset sizes continue to grow, the computational cost of training dense models becomes prohibitive. A promising approach to scaling model capacity while controlling computational costs is the Mixture of Experts (MoE) architecture. Rather than activating all parameters for every token, MoE architectures [29, 70] replace dense Feed-Forward Network (FFN) layers with multiple expert networks and a gating mechanism that routes each token to a subset of experts. This enables dramatic parameter scaling, as demonstrated by DeepSeek-V3 with 671B parameters but 37B active per token [28], while keeping computational costs manageable.

Still, the training of these LLMs presents unprecedented computational challenges. Modern models require millions of GPU-hours to train [3, 5], necessitating sophisticated distributed training strategies across hundreds or thousands of accelerators [85]. The memory footprint of these models far exceeds the capacity of individual GPUs; for instance, a 175-billion-parameter model in mixed precision (FP16) requires approximately 350 GB just to store its parameters, before accounting for activations, gradients, and optimizer states, which can multiply memory requirements by a factor of 12-16 [65].

These constraints have driven the development of sophisticated parallelization techniques that partition both the model and training data across distributed systems, distributing memory and computational requirements across hundreds or thousands of accelerators. The efficiency of these strategies depends critically on the underlying communication infrastructure, as synchronization overhead can quickly dominate training time at scale.

The relationship between model size, dataset size, and performance is characterized by scaling laws [37, 44], which provide guidance for efficient resource allocation during training. These laws reveal that optimal training requires not just large models, but also proportionally large datasets and compute budgets.

Meeting these computational demands necessitates specialized hardware infrastructure purpose-built for the unique requirements of LLM training. The following section describes the GPU clusters, interconnect technologies, and node architectures that enable distributed training at the scale required by modern LLMs.

2.2 HARDWARE INFRASTRUCTURE FOR LLM TRAINING

Modern LLM training clusters are purpose-built supercomputers designed to maximize GPU utilization and training throughput. A typical training cluster consists of hundreds to thousands of GPU nodes connected through high-performance networks in carefully designed topologies [42, 54].

2.2.1 GPU Accelerators

The training of large language models relies almost exclusively on GPUs as the primary computational accelerator. Modern GPUs are specifically designed for the massive parallelism required in deep learning workloads, offering thousands of cores capable of executing operations simultaneously. NVIDIA has dominated the LLM training landscape with its data center GPU families, particularly the Ampere [8], Hopper [9], Blackwell [11], and most recently Rubin [22] architectures.

The NVIDIA A100 GPU features 80 GB of High-Bandwidth Memory (HBM) with 2 TB/s memory bandwidth and delivers up to 312 Tera-FLOPS of FP16 performance with Tensor Cores [8]. The H100 provides substantial improvements with 80 GB of HBM memory, 3.35 TB/s memory bandwidth, and up to 1,979 Tera-FLOPS of FP16 performance [9]. These Tensor Cores are specialized matrix multiplication units optimized for the mixed-precision arithmetic commonly used in neural network training [53].

Beyond NVIDIA, alternative accelerators have emerged. Google's Tensor Processing Units (TPUs), particularly the TPU v4 and v5 generations [43, 52], are custom Application-Specific Integrated Circuits (ASICs) designed specifically for machine learning workloads. AMD offers the MI250X and MI300 series GPUs [1, 2], while Intel has developed the Habana Gaudi processors [7]. However, NVIDIA's mature software ecosystem, particularly CUDA and associated libraries, maintains its dominant position in LLM training [68].

2.2.2 Node Configuration

Individual compute nodes form the building blocks of training clusters, with each node designed to maximize local computation and communication efficiency. Standard configurations include 8 GPUs per node, as present in NVIDIA DGX systems [13, 20, 21], connected via NVLink and NVSwitch for maximum intra-node bandwidth. Each node typically contains dual high-core-count Central Processing Units (CPUs), several hundred Gigabytes to over one Terabyte of system

memory, and fast Non-Volatile Memory Express (NVMe) storage for dataset access [54]. Power consumption per node can exceed 10 kW for H100-based systems, necessitating advanced cooling infrastructure [9].

2.2.3 Interconnect Technologies

The performance hierarchy of LLM training clusters is determined by two distinct interconnect layers: intra-node connections between GPUs within a server, and inter-node networking between servers. These layers exhibit vastly different bandwidth and latency characteristics, directly impacting the efficiency of different parallelism strategies.

intra-node interconnects Within a single server node, NVIDIA's NVLink provides direct GPU-to-GPU communication with significantly higher bandwidth than traditional Peripheral Component Interconnect Express (PCIe) connections. The third generation NVLink in A100 GPUs offers 600 GB/s bidirectional bandwidth per GPU [8], while NVLink 4.0 in H100 GPUs reaches 900 GB/s [9]. For systems with eight GPUs per node, NVSwitch provides full non-blocking connectivity, enabling any GPU to communicate with any other at full NVLink speed simultaneously [30].

inter-node interconnects Connecting multiple nodes requires high-performance networking fabrics capable of sustaining the aggregate bandwidth demands of hundreds of concurrent GPU-to-GPU communications. InfiniBand has become the de facto standard for High-Performance Computing (HPC) and AI clusters, with HDR InfiniBand providing 200 Gigabit per Second (Gbps) (25 GB/s) per port and NDR InfiniBand reaching 400 Gbps (50 GB/s) [18, 76, 80]. NVIDIA's acquisition of Mellanox has led to tight integration between GPUs and InfiniBand through GPU-Direct Remote Direct Memory Access (RDMA), allowing direct memory access between GPUs across nodes without CPU involvement [69].

The bandwidth disparity between intra-node (600-900 GB/s per GPU) and inter-node (25-50 GB/s per port) interconnects often exceeding a factor of 10, fundamentally shapes distributed training strategy. Parallelism strategies that require frequent cross-node communication incur substantial performance penalties, motivating the careful orchestration of parallelism strategies within node boundaries.

However, possessing high-bandwidth interconnects alone is insufficient. Efficient utilization requires specialized software libraries that can exploit these hardware capabilities while managing the complexity of collective communication patterns across hundreds or thousands of devices. This software layer, which abstracts the underlying hardware

topology and implements optimized communication algorithms, is provided by the NVIDIA Collective Communications Library (NCCL).

2.3 NCCL: GPU COLLECTIVE COMMUNICATION LIBRARY

All collective communication operations in modern distributed training frameworks are implemented through specialized libraries optimized for GPU-to-GPU transfers. While each major hardware vendor provides its own collective communication library, NCCL has emerged as the de facto standard for NVIDIA-based GPU clusters, offering highly optimized implementations tailored to deep learning communication patterns [24]. As this work focuses on NVIDIA GPU systems, understanding NCCL's design and operation is essential for interpreting the communication patterns analyzed in this thesis.

2.3.1 Design Principles and Architecture

NCCL operates entirely within GPU memory space, avoiding unnecessary host memory copies and enabling direct device-to-device transfers [24]. The library integrates with GPU-direct technologies such as GPU-direct RDMA for inter-node communication and GPU-direct Peer-to-Peer for intra-node transfers to enable zero-copy data movement [69].

Recent comprehensive analysis by Hu et al. [38] reveals NCCL's internal architecture employs a channel-based execution model. Each channel represents a persistent execution context bound to GPU streaming multiprocessors, with collective operations decomposed into work units distributed across multiple channels. The implementation maintains circular buffers (typically 8 slots) per channel to pipeline data chunks, allowing communication, computation, and memory movement to proceed concurrently [38].

2.3.2 Collective Operations

NCCL implements the core collective primitives used in distributed training, as shown in Figure 2.1:

- **Broadcast**: Distributes data from a single root device to all devices.
- **AllGather**: Collects data from all devices and concatenates them, distributing the complete concatenated result to every device.
- **Reduce**: Performs element-wise reduction across inputs and delivers the result only to the root device.

- **AllReduce**: Performs an element-wise reduction, typically summation, across all participating devices and distributes the reduced result back to all devices. If device i contributes tensor x_i, all devices receive $y = \text{reduce}(x_0, x_1, \ldots, x_{n-1})$.
- **ReduceScatter**: Performs element-wise reduction and distributes different chunks of the reduced result to different devices.

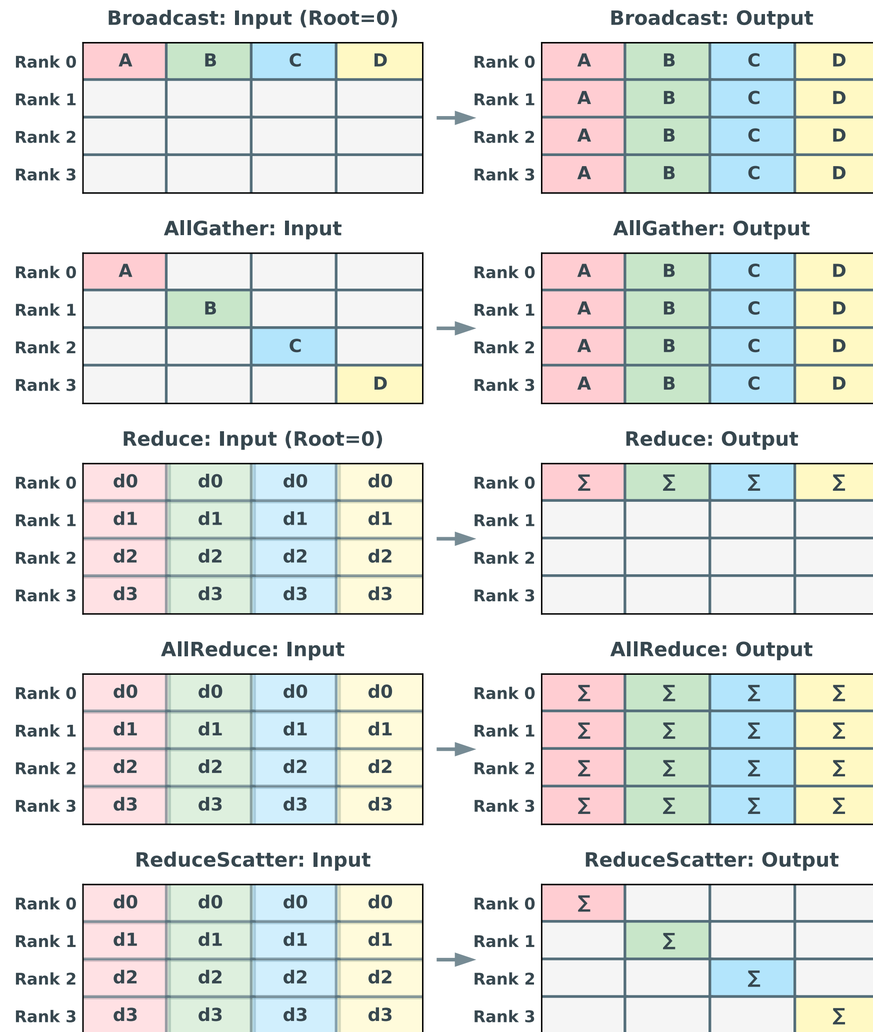


Figure 2.1: Visualization of NCCL collective operations across four ranks. Each row depicts a different collective, showing the input memory layout on the left and the resulting output distribution on the right. Colors represent data chunks, and empty slots indicate unused memory.

2.3.2.1 *Algorithm Selection*

NCCL implements multiple algorithms for each collective operation and selects dynamically based on message size and topology [24, 38].

For `AllReduce`, the ring algorithm arranges devices in a logical ring where each communicates with one predecessor and successor [59]. The algorithm proceeds through `ReduceScatter` (input data partitioned into n chunks circulates and progressively reduces) followed by `AllGather` (reduced chunks circulate to distribute complete results). This achieves bandwidth optimality as all network links operate simultaneously, with time complexity:

$$T_{\text{ring}} \approx \frac{2S(n-1)}{nB} \tag{2.2}$$

where S is message size, n is device count, and B is link bandwidth. As n increases, this approaches the theoretical optimum $\frac{2S}{B}$ [59].

For smaller messages where latency dominates, NCCL employs tree-based reduction [61]. NCCL implements double binary trees with distinct structures for reduction and broadcast phases executing concurrently [38]. This approach completes in $2\log_2(n)$ steps, providing lower latency than ring algorithms for small transfers.

NCCL dynamically selects between ring and tree algorithms through runtime heuristics, typically employing tree-based approaches for messages below 1-2 Megabyte (MB) and ring-based approaches for larger transfers [38].

2.3.3 Protocol Variants and Topology Awareness

NCCL implements three communication protocols selected dynamically based on message characteristics [38]:

- **Simple Protocol:** Targets large messages, performing direct transfers with minimal per-element synchronization overhead, optimized for bandwidth-intensive operations.
- **Low Latency (LL) Protocol:** Serves small messages, employing fine-grained synchronization at individual data element granularity, reducing latency at the cost of increased synchronization overhead.
- **LL128 Protocol:** Enhances LL by processing 128-bit chunks per synchronization, balancing between LL's latency characteristics and Simple's bandwidth efficiency.

NCCL performance depends critically on exploiting physical network topology. During initialization, NCCL performs topology discovery by detecting NVLink connectivity, PCIe hierarchy, and network

fabric arrangements [24]. Based on this information, the library constructs optimized communication graphs prioritizing high-bandwidth links and avoiding congestion. In systems where NVLink provides 900 GB/s connectivity while PCIe offers 64 GB/s, NCCL constructs ring topologies maximizing NVLink utilization. In multi-node configurations, the library coordinates intra-node NVLink communication with inter-node InfiniBand transfers, carefully scheduling operations to minimize contention [24].

2.3.4 Integration

While NCCL provides the low-level communication primitives, the question remains: how should these primitives be orchestrated to efficiently train models that exceed the memory capacity of individual devices? This challenge is addressed by distributed training frameworks, which implement sophisticated parallelization strategies that partition computation and data across the GPU cluster. These strategies determine which NCCL collectives are invoked, when they execute, and how frequently, ultimately shaping the communication patterns that this thesis aims to analyze.

Modern training frameworks rely exclusively on NCCL for GPU collective communication on NVIDIA hardware. PyTorch's distributed package uses NCCL as its default backend for multi-GPU training, while frameworks like Megatron-LM [54, 71] build parallelization strategies atop NCCL's collective primitives. This standardization enables consistent communication performance across the ecosystem.

2.4 DISTRIBUTED TRAINING STRATEGIES AND COMMUNICATION PATTERNS

Scaling transformer training beyond single-device capabilities requires distributing both computation and model state across multiple accelerators. To address the ever-increasing computational requirements of modern LLMs, researchers and practitioners have developed sophisticated distributed training frameworks that combine multiple parallelization strategies [54, 66, 88]. These frameworks must balance competing objectives: maximizing computational throughput, minimizing memory consumption per device, and reducing communication overhead.

This section examines the primary parallelization approaches employed by modern training frameworks and the communication patterns they induce, focusing on their concrete implementation in Megatron-LM. While the theoretical communication costs of these strategies are well understood [54, 71], the practical implications of their network

traffic signatures including operation timing, sequencing, and efficiency, remain underexplored.

2.4.1 Data Parallelism

Data Parallelism (DP) represents the most straightforward approach to distributed training: identical copies of the complete model are deployed across multiple devices, with each device processing a distinct portion of the training batch [33]. This strategy scales the effective batch size linearly with device count, enabling faster training through increased sample throughput.

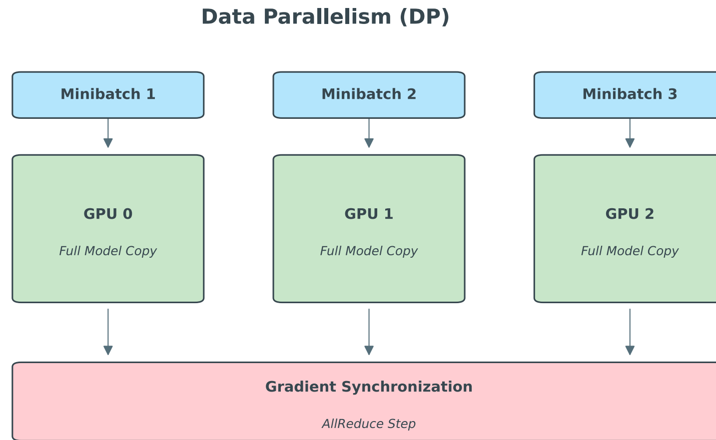


Figure 2.2: Data parallelism distributes training data across devices while maintaining identical model copies. Each device processes a different minibatch independently, computing local gradients that are synchronized via `AllReduce` before parameter updates.

During each training iteration, devices independently execute forward and backward passes on their assigned data partitions, producing gradient tensors specific to their local batches, see Figure 2.2. To maintain consistency across model replicas, these locally computed gradients must be aggregated before parameter updates. The aggregation employs an `AllReduce` collective operation that computes element-wise summation of gradient tensors across all devices [77]. If device i computes gradients g_i, the `AllReduce` ensures every device receives:

$$g_{\text{synchronized}} = \frac{1}{n_{\text{DP}}} \sum_{i=1}^{n_{\text{DP}}} g_i \tag{2.3}$$

where n_{DP} denotes the data-parallel group size. Following synchronization, all devices apply identical parameter updates, preserving model equivalence across replicas.

COMMUNICATION VOLUME The communication volume for DP in Megatron-LM is dominated by the synchronization of gradients across data-parallel ranks during training. Let P denote the total number of model parameters and let β represent the number of bytes per parameter element. For a transformer model with l layers, hidden dimension h, vocabulary size V, and sequence length s, the total parameter count can be approximated as

$$P = 12lh^2 \left(1 + \frac{1}{12h} + \frac{1}{12lh}\frac{V+s}{h}\right) \tag{2.4}$$

which accounts for the dominant contributions from attention projections, feed-forward networks, layer normalization, and embedding parameters. For a ring-based `AllReduce` implementation, the communication volume per rank is given by $2 \cdot \frac{n_{\mathrm{DP}}-1}{n_{\mathrm{DP}}} \cdot P \cdot \beta$, reflecting the combined `ReduceScatter` and `AllGather` phases. Because Megatron-LM performs a single gradient synchronization per training iteration after gradient accumulation, this volume directly characterizes the DP communication cost per iteration. For sufficiently large data-parallel world sizes, the factor $\frac{n_{\mathrm{DP}}-1}{n_{\mathrm{DP}}}$ approaches unity, allowing the DP communication volume to be approximated as:

$$V_{\mathrm{DP}} \approx 2 \cdot P \cdot \beta. \tag{2.5}$$

When DP is combined with TP and PP, gradient synchronization occurs only over the locally owned parameter shards within each tensor-parallel group and pipeline stage. In this case, the effective parameter count participating in DP communication is reduced to $\frac{P}{n_{\mathrm{TP}} \cdot n_{\mathrm{PP}}}$. Accordingly, the generalized DP communication volume in Megatron-LM can be expressed as:

$$V_{\mathrm{DP}} = 2 \cdot \frac{n_{\mathrm{DP}} - 1}{n_{\mathrm{DP}}} \cdot \frac{P}{n_{\mathrm{TP}} \cdot n_{\mathrm{PP}}} \cdot \beta. \tag{2.6}$$

This formulation captures the dominant communication cost of DP and serves as a baseline against which more complex parallelism strategies can be compared.

TEMPORAL CHARACTERISTICS AND OPTIMIZATIONS A key advantage of DP is its temporal sparsity: gradient synchronization occurs once per training step after the backward pass completes. Modern frameworks exploit this by overlapping communication with computation: as gradients for later layers become available during backpropagation, they are immediately enqueued for reduction while earlier layers continue computing [46, 86].

Memory-efficient variants like Zero Redundancy Optimizer (ZeRO) [65] and Fully Sharded Data Parallel (FSDP) [88] partition optimizer states, gradients, and parameters across data-parallel groups. These techniques replace single `AllReduce` operations with `ReduceScatter`

and `AllGather` sequences, increasing communication volume by approximately 50% while enabling larger model training within memory constraints.

2.4.2 Tensor Parallelism

When individual model layers exceed single-device memory capacity, Tensor Parallelism (TP) becomes necessary. Introduced by Shoeybi et al. [71], TP partitions weight matrices and distributes matrix operations across multiple devices. Rather than replicating computation as in DP, TP distributes computation across devices that collectively execute each layer.

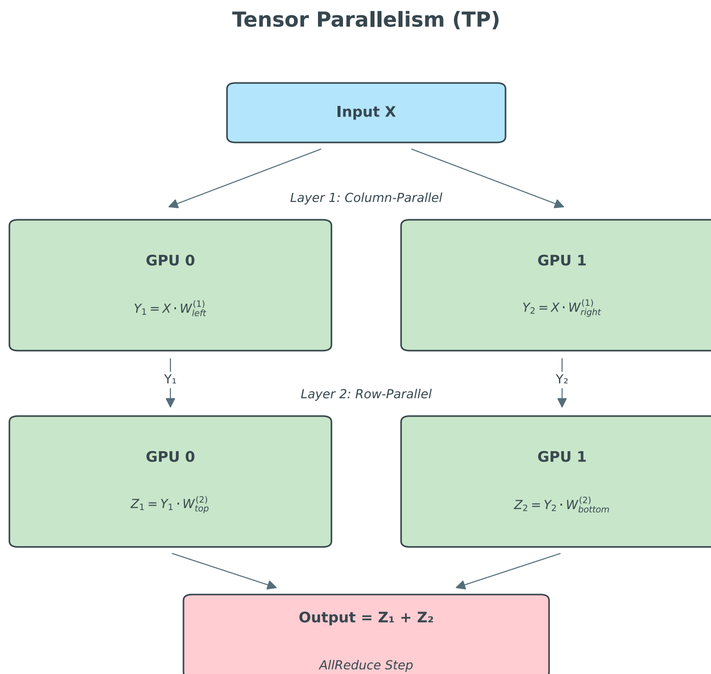


Figure 2.3: Tensor parallelism as implemented in Megatron-LM. The first layer uses column-parallel partitioning of weight matrix $W^{(1)}$, allowing independent computation of partial activations Y_1 and Y_2. These feed directly into the second layer with row-parallel partitioning of $W^{(2)}$, requiring only a single `AllReduce` operation to aggregate final outputs Z_1 and Z_2. This strategic arrangement minimizes communication overhead.

MATRIX PARTITIONING STRATEGY Consider a transformer layer's FFN with two sequential matrix multiplications. TP partitions the first weight matrix column-wise across n_{TP} devices, enabling each device to compute a portion of intermediate activations independently. The non-linearity is applied locally without communication. The second

weight matrix is partitioned row-wise, allowing each device to compute partial outputs from its local activations. This strategic placement, column-wise then row-wise partitioning, enables non-linearity computation without synchronization and requires only a single `AllReduce` operation per two-layer block [71], see Figure 2.3.

COMMUNICATION VOLUME TP partitions the model's weight matrices across multiple devices and requires frequent synchronization of partial results during execution. In Megatron-LM, each transformer layer performs two `AllReduce` operations in the forward pass and two `AllReduce` operations in the backward pass to combine tensor-parallel shards. The FFN requires one `AllReduce` operation after the second linear transformation to aggregate partial outputs, occurring in both forward and backward passes. The multi-head attention mechanism similarly requires synchronization after computing attention outputs. These collectives operate on activation tensors produced by linear layers whose outputs are distributed across the tensor-parallel group.

Let L_{stage} denote the number of transformer layers assigned to a pipeline stage, b the microbatch size, s the sequence length, h the hidden dimension, n_{TP} the tensor-parallel world size, and β the number of bytes per tensor element. For a ring-based `AllReduce` implementation, the communication scaling factor is $\frac{n_{\text{TP}}-1}{n_{\text{TP}}}$. Accounting for four `AllReduce` operations per layer, two in forward pass, two in backward pass, and aggregating across all layers within a pipeline stage, the total tensor-parallel communication volume per device and per microbatch is given by:

$$V_{\text{TP}} = L_{\text{stage}} \cdot \left(8 \cdot b \cdot s \cdot h \cdot \frac{n_{\text{TP}} - 1}{n_{\text{TP}}}\right) \cdot \beta \tag{2.7}$$

[54] This expression highlights that TP communication volume increases with tensor-parallel size, motivating its restriction to high-bandwidth, low-latency interconnects such as NVLink within a single node.

The high frequency of TP communication, multiple times per layer in both forward and backward passes, makes this strategy highly sensitive to interconnect performance. Consequently, TP is typically constrained to intra-node communication where high-bandwidth NVLink (600-900 GB/s) is available [8, 9]. Extending TP across nodes via InfiniBand (25-50 GB/s) introduces prohibitive overhead given per-layer synchronization requirements [54].

2.4.3 Pipeline Parallelism

Pipeline Parallelism (PP) addresses model size through vertical partitioning: consecutive layers are distributed across different devices, forming a sequential processing pipeline [39]. Unlike TP's horizontal

partitioning within layers, PP creates explicit inter-device dependencies where stage i must complete before stage $i + 1$ can proceed.

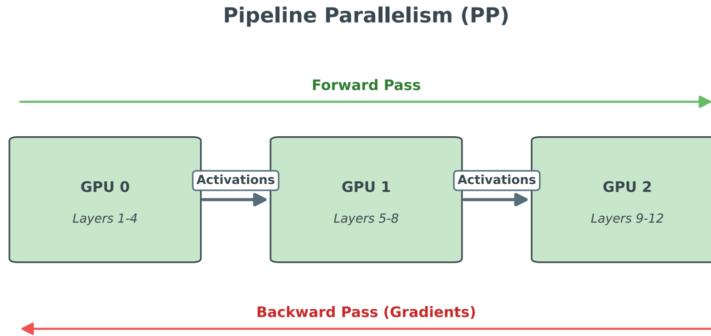


Figure 2.4: Pipeline parallelism partitions the model vertically, assigning consecutive layers to different GPUs. Activation tensors flow forward through pipeline stages during the forward pass, while gradients flow backward during backpropagation.

MICROBATCHING AND SCHEDULING In a PP configuration with n_{PP} stages, the model's L layers are divided into n_{PP} contiguous groups assigned to separate devices, see Figure 2.4. To avoid severe underutilization, the training batch is subdivided into $m = \frac{B}{b \cdot n_{\text{DP}}}$ microbatches (where B is global batch size, b is microbatch size) that flow through pipeline stages in overlapped fashion. The 1F1B (one forward, one backward) schedule [54], see Figure 2.5, alternates forward and backward passes for different microbatches on each stage, maximizing occupancy and minimizing idle time.

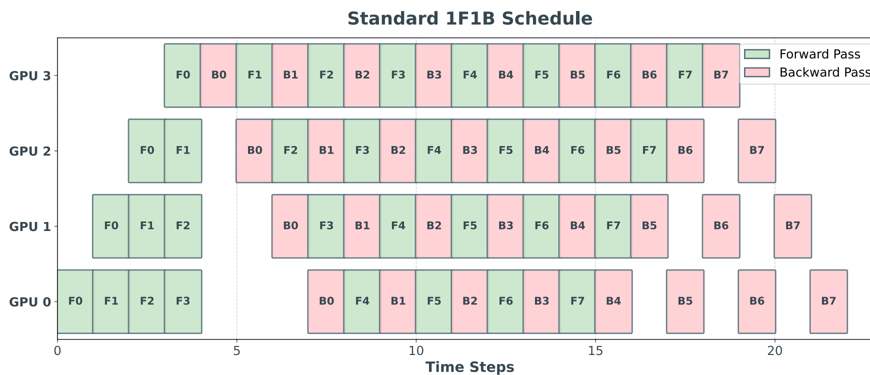


Figure 2.5: Standard one-forward-one-backward (1F1B) pipeline parallel schedule. After an initial warm-up phase, pipeline stages alternate between forward and backward passes on different microbatches, enabling overlap of computation across stages and reducing pipeline idle time. A final drain phase completes the remaining backward passes.

COMMUNICATION VOLUME PP partitions the model into sequential stages distributed across devices and relies on point-to-point communication of activations between adjacent stages. Unlike DP and TP which rely on collective operations, PP employs point-to-point `Send` and `Recv` primitives for inter-stage communication. During forward propagation, stage i sends activation tensors to stage $i+1$ upon completing computation; during backpropagation, stage $i+1$ sends gradient tensors back to stage i. These transfers occur only between adjacent stages, creating a fundamentally different communication pattern from collectives involving all devices simultaneously.

During the forward pass, each stage transmits its output activations to the next stage, while during the backward pass, gradients with respect to these activations are sent in the reverse direction. Let b denote the microbatch size, s the sequence length, h the hidden dimension, and β the number of bytes per tensor element. In the standard pipeline configuration, the communication volume exchanged between a pair of consecutive stages for either the forward or backward pass per microbatch is:

$$V_{\text{PP}} = b \cdot s \cdot h \cdot \beta \tag{2.8}$$

[54] When TP is enabled, Megatron-LM employs scatter and gather optimizations such that each tensor-parallel rank communicates only its local shard of the activation tensor. This optimization reduces the communicated volume per rank to:

$$V_{\text{PP,TP}} = b \cdot s \cdot \frac{h}{n_{\text{TP}}} \cdot \beta \tag{2.9}$$

[54] thereby lowering inter-stage traffic over inter-node links.

PIPELINE EFFICIENCY AND BUBBLE OVERHEAD Point-to-point communication enables effective overlap with computation. While one microbatch transfers between stages, other microbatches actively compute on sending and receiving stages. This overlap, combined with relatively infrequent communication, once per microbatch per boundary versus multiple times per layer in TP, makes PP communication overhead manageable.

However, in addition to communication overhead, pipeline parallelism introduces pipeline bubbles due to stage imbalance and pipeline fill and drain effects. If n_{PP} denotes the number of pipeline stages and m the number of microbatches per batch, the fraction of ideal execution time lost to pipeline bubbles is given by:

$$\text{Bubble Fraction} = \frac{n_{\text{PP}} - 1}{m} \tag{2.10}$$

This indicates that increasing the number of microbatches m reduces bubble overhead, though at the cost of increased memory consumption

for storing intermediate activations. PP proves particularly effective for inter-node parallelization where collective operation latency becomes prohibitive [54].

2.4.4 Sequence and Context Parallelism

SEQUENCE PARALLELISM Sequence Parallelism (SP) addresses the challenge of training transformers with extremely long input sequences by partitioning the sequence dimension across multiple devices [48]. An input sequence of length n is divided into n_{SP} segments, with each device processing n/n_{SP} consecutive tokens. This distributes both the computation and memory of attention ($\mathcal{O}(n^2)$) across devices, while position-wise operations like FFNs proceed independently on each segment. SP requires `AlltoAll` operations during attention to exchange key-value pairs and `AllReduce` operations for layer normalization to compute statistics globally across the full sequence.

CONTEXT PARALLELISM Context Parallelism (CP) is a specialized sequence partitioning technique designed to reduce activation memory consumption when training with long contexts [12]. While conceptually similar to SP, the critical distinction lies in the communication strategy: CP uses `AllGather` to assemble the complete key-value sequence on each device before computing attention, whereas SP uses `AlltoAll` to enable distributed attention computation. This means each device processes only its local query segment but attends to the full context. In the backward pass, `ReduceScatter` distributes gradients back to their originating devices. CP reduces both computation and activation memory by a factor of n_{CP} and is most effective for sequences of 8K tokens or longer [12].

2.4.5 Expert Parallelism

Expert Parallelism (EP) is specifically designed for MoE models, where feed-forward layers are replaced with multiple expert networks and a gating mechanism routes tokens to subsets of experts [29, 70]. EP distributes these experts across devices, enabling models with dramatically larger parameter counts while maintaining manageable per-token computational costs.

EXPERT DISTRIBUTION AND TOKEN ROUTING In MoE transformers, each layer contains multiple expert networks. A learned gating function determines which experts process each token, with common configurations activating 1-2 experts per token. EP partitions experts across n_{EP} devices such that each device hosts a subset of experts [14, 50], see Figure 2.6. For example, with 64 experts and $n_{\text{EP}} = 8$, each device hosts 8 experts.

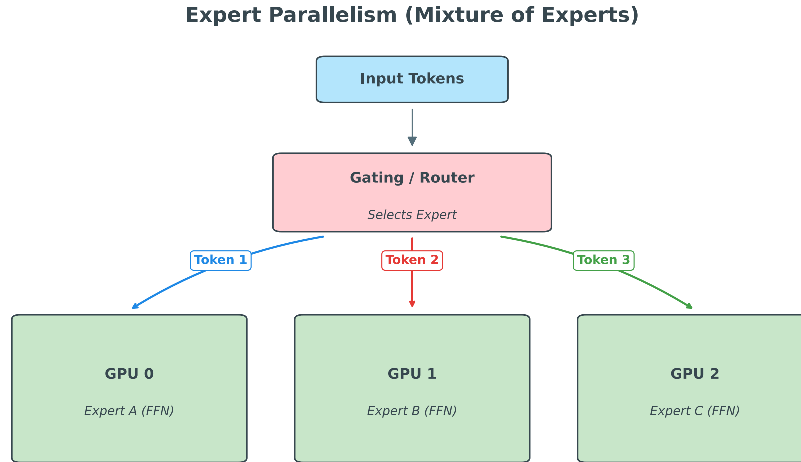


Figure 2.6: Expert parallelism for Mixture of Experts architectures. A gating mechanism dynamically routes input tokens to specialized expert networks distributed across GPUs. Each token is processed by its selected expert, with `AlltoAll` communication coordinating token routing and result aggregation.

COMMUNICATION PATTERN EP introduces irregular communication patterns due to dynamic token routing. Two primary token dispatching strategies are available for EP communication:

The `AlltoAll` dispatcher is the standard approach for EP configurations with $n_{\text{EP}} > 1$. Tokens are routed to the devices hosting their selected experts through an `AlltoAll` communication operation that exchanges tokens between all devices in the EP group. After expert computation, results are returned to their originating devices for aggregation through another `AlltoAll` operation, with analogous operations during backpropagation for gradient flow. This strategy enables efficient load distribution across devices by physically moving tokens to their assigned experts.

The `AllGather` dispatcher provides an alternative where all tokens are gathered to each GPU, eliminating inter-GPU token movement. Instead of routing tokens to specific devices, each device receives a complete copy of all tokens and selects only those assigned to its local experts. This approach is particularly beneficial for TP-only setups or small EP configurations. While `AllGather` increases total data transfer by replicating all tokens across devices, it can reduce latency and simplify load balancing in certain configurations [50].

COMMUNICATION VOLUME FOR ALLTOALL-BASED EXPERT PARALLELISM The derivation of the communication volume for EP centers on the `AlltoAll` collective operations required for token routing across a distributed system [50]. In a standard MoE layer utilizing top-k gating, each of the $L = B \times S$ tokens is assigned to k experts based on

gating probabilities. Here, B represents the batch size, which may encompass multiple microbatches, S is the sequence length, and k denotes the number of experts activated per token. Under the assumption of uniform routing across an expert-parallel world size of n_{EP}, the fraction of tokens requiring transfer to a remote rank is given by $f = 1 - \frac{1}{n_{\text{EP}}}$. During the token dispatching phase, these remotely routed tokens are communicated as hidden representations of dimensionality h via an `AlltoAll` operation to ensure that each rank receives the tokens corresponding to its locally hosted experts. A second `AlltoAll` operation is required during the token restore phase to return the processed expert outputs to their original ranks, preserving the sequential alignment of the transformer pipeline. Because the backward pass mirrors the forward routing and restoration steps, a total of four major communication events occur per training iteration. Consequently, the total communication volume for expert parallelism can be approximated as:

$$V_{\text{EP,total}} \approx 4 \cdot B \cdot S \cdot k \cdot h \cdot \left(1 - \frac{1}{n_{\text{EP}}}\right) \tag{2.11}$$

This approximation is robust for standard EP configurations. In practice, the realized communication volume may be inflated by the use of a capacity factor $\text{CF} > 1$, which pads routing buffers to accommodate load imbalance across experts. Moreover, fine-grained MoE architectures inherently increase communication by activating a larger number of experts per token, corresponding to larger values of k. The introduction of TP further augments the communication footprint through additional `AllGather` and `ReduceScatter` collectives within expert groups.

A critical challenge in EP is load imbalance: if the gating function routes tokens unevenly across experts, some devices become overloaded while others remain underutilized. MoE training typically employs auxiliary loss terms to encourage balanced expert utilization [27, 29]. Note that when combining EP with TP, SP must be enabled for proper coordination [23].

2.4.6 Multidimensional Parallelism

Modern large-scale training employs multiple parallelization strategies simultaneously, termed multidimensional or N-dimensional parallelism [54]. The most common configuration, 3D parallelism, combines DP, TP, and PP. More complex models may additionally employ CP for long sequences and EP for MoE architectures:

$$n_{\text{total}} = n_{\text{DP}} \times n_{\text{TP}} \times n_{\text{PP}} \times n_{\text{CP}} \times n_{\text{EP}} \tag{2.12}$$

Each parallelization strategy operates on disjoint device groups with distinct communication characteristics:

- **DP replicas:** Span the cluster, synchronize gradients through `AllReduce` operations across all replicas.
- **TP groups:** Typically co-located within nodes, perform synchronized computation using `AllReduce` operations to combine partial results from partitioned layers.
- **PP groups:** Form chains of stages potentially spanning nodes, exchange activations and gradients via point-to-point `Send`/`Recv` operations.
- **CP groups:** Share sequence segments using `AllGather` operations to assemble complete key-value pairs for attention computation.
- **EP groups:** Host different expert subsets, use `AlltoAll` operations for token routing between experts.

COMMUNICATION OVERLAP AND OPTIMIZATION EFFECTS Communication patterns from different strategies overlap both temporally and spatially. Within a training iteration, TP `AllReduce` occurs during layer computation, PP point-to-point transfers happen between microbatch processing, CP operations occur at attention layers, and DP gradient `AllReduce` overlaps with backpropagation. Frameworks carefully schedule these communications to maximize overlap with useful work [54].

Different communication patterns utilize distinct network resources: TP `AllReduce` primarily consumes intra-node NVLink bandwidth, PP point-to-point uses inter-node InfiniBand, while DP `AllReduce` spans the entire hierarchy. When strategies are combined, they can create synergistic optimization opportunities. For instance, when TP is active within pipeline stages, the activation volume transferred between stages distributes across tensor-parallel devices, consistent with the scatter and gather optimizations described in Section 2.4.3:

$$V_{\text{PP,TP}} = b \cdot s \cdot \frac{h}{n_{\text{TP}}} \cdot \beta \text{ per device} \tag{2.13}$$

This reduction in per-device communication volume is one reason why combining TP and PP can be effective: TP reduces the PP boundary transfer size by a factor of n_{TP}, while PP enables training models too large even for tensor-parallel configurations. However, this benefit must be balanced against TP's requirement for high-bandwidth intra-node connectivity and PP's introduction of pipeline bubbles.

CONFIGURATION TRADE-OFFS Selecting appropriate parallelization degrees involves balancing competing factors:

- Increasing n_{DP} improves batch throughput but may degrade optimization quality with excessive global batch sizes.

- Increasing n_{TP} reduces per-device memory but increases communication frequency and sensitivity to interconnect performance.
- Increasing n_{PP} enables larger models but introduces pipeline bubbles with idle time scaling as $\frac{1}{m}$.
- Increasing n_{CP} enables longer sequences but adds attention-specific communication overhead.
- Increasing n_{EP} supports more experts but creates irregular, load-dependent communication patterns.

Empirical studies demonstrate that suboptimal parallelization combinations can result in up to 2× lower throughput even with identical hardware [54]. The practical challenge lies not only in understanding these trade-offs theoretically, but in diagnosing how specific configurations manifest as network traffic patterns during actual training runs.

To perform such diagnosis, practitioners must correlate the semantically rich NCCL debug logs with hardware profiler traces from Nsys that capture nanosecond-accurate kernel execution timing. Bridging this gap requires aligning these two fundamentally mismatched data sources despite missing entries, reordering, and granularity differences.

The Needleman-Wunsch algorithm from computational biology provides a solution to this alignment problem, adapting techniques originally designed for protein sequence comparison to correlate distributed training traces.

2.5 THE NEEDLEMAN-WUNSCH ALGORITHM

The need to correlate semantically rich but timing-imprecise NCCL debug logs with timing-precise but semantically sparse Nsys kernel traces requires a robust method for matching sequences of events that may have undergone insertions, deletions, or reordering. This is fundamentally a sequence alignment problem, analogous to those encountered in computational biology when comparing protein or DNA sequences.

The Needleman-Wunsch algorithm, proposed by Saul B. Needleman and Christian D. Wunsch in 1970 [55], provides a principled solution to this class of problems. Originally developed for comparing protein sequences to infer evolutionary relationships, the algorithm has found applications far beyond its initial domain, including DNA sequence analysis, speech recognition, and, as leveraged in this thesis, temporal trace alignment for distributed systems profiling.

The fundamental problem addressed by Needleman-Wunsch is: given two sequences of symbols, find the optimal global alignment

that maximizes similarity while accounting for insertions, deletions, and substitutions. Unlike local alignment algorithms such as Smith-Waterman [74], which identify regions of high similarity within sequences, Needleman-Wunsch performs end-to-end alignment, making it particularly suitable for comparing sequences of similar length and overall structure, precisely the scenario encountered when matching NCCL operation logs to Nsys kernel executions.

2.5.1 Problem Formulation

SEQUENCE DEFINITIONS Consider two sequences to be aligned:

- Sequence A of length n: $A = a_1 a_2 a_3 \dots a_n$, where each $a_i \in \Sigma$
- Sequence B of length m: $B = b_1 b_2 b_3 \dots b_m$, where each $b_j \in \Sigma$

For biological sequences, Σ represents amino acids or nucleotides. In the context of trace alignment for this thesis, Σ represents the set of possible communication operations, e.g., NCCL collectives like `AllReduce`, `AllGather`, `ReduceScatter`, `Send`, `Recv`.

ALIGNMENT REPRESENTATION An alignment between sequences A and B is a pair of strings A' and B' of equal length L (where $L \geq \max(n, m)$) formed by inserting gap characters ("$-$") into A and B such that when gaps are removed, the original sequences are recovered. For example, aligning $A =$ `GCAT` and $B =$ `GATT` might produce:

$$A' = \texttt{G C A T -}$$
$$B' = \texttt{G - A T T}$$

SCORING SYSTEM The quality of an alignment is quantified by a scoring function consisting of:

1. **Substitution Matrix** $S(x, y)$: Returns the score for aligning character x with character y.
   - *Match score:* $S(x, x) = S_{match}$ (typically positive, e.g., $+1$, $+5$)
   - *Mismatch penalty:* $S(x, y) = S_{mismatch}$ for $x \neq y$ (typically negative, e.g., -1, -3)

   For protein sequences, sophisticated matrices like BLOSUM [36] or PAM [25] are used, derived from observed evolutionary substitution frequencies.

2. **Gap Penalty** d: A penalty, typically also negative, for introducing a gap. The standard Needleman-Wunsch algorithm uses linear gap penalties, where each gap incurs a fixed penalty d.

Extensions like Gotoh's algorithm [32] handle affine gap penalties (separate opening and extension penalties) to reflect that extending an existing gap is more likely than opening a new one.

The objective is to find the alignment (A', B') that maximizes the total score:

$$\text{Score}(A', B') = \sum_{i=1}^{L} \text{score}(A'_i, B'_i) \tag{2.14}$$

where $\text{score}(A'_i, B'_i) = S(A'_i, B'_i)$ if neither is a gap, and d if exactly one is a gap.

2.5.2 Dynamic Programming Solution

The Needleman-Wunsch algorithm employs dynamic programming to efficiently solve what would otherwise be computationally intractable. The algorithm exploits the principle of optimality: an optimal alignment of sequences A and B must contain optimal alignments of their prefixes.

A scoring matrix F of dimensions $(n+1) \times (m+1)$ is defined, where $F(i, j)$ represents the maximum alignment score for the prefixes $A[1..i]$ and $B[1..j]$.

PHASE 1: INITIALIZATION The first row and column represent aligning one sequence against an empty sequence (all gaps):

$$F(0, 0) = 0 \tag{2.15}$$

$$F(i, 0) = i \cdot d, \quad \text{for } 1 \leq i \leq n \tag{2.16}$$

$$F(0, j) = j \cdot d, \quad \text{for } 1 \leq j \leq m \tag{2.17}$$

PHASE 2: MATRIX FILLING For each cell (i, j) where $1 \leq i \leq n$ and $1 \leq j \leq m$, the score is computed as:

$$F(i, j) = \max \begin{cases} F(i-1, j-1) + S(a_i, b_j) & \text{(Diagonal: match/mismatch)} \\ F(i-1, j) + d & \text{(Vertical: gap in } B\text{)} \\ F(i, j-1) + d & \text{(Horizontal: gap in } A\text{)} \end{cases} \tag{2.18}$$

Each case corresponds to a different alignment decision:

- **Diagonal:** Align a_i with b_j (match if $a_i = b_j$, mismatch otherwise)
- **Vertical:** Insert a gap in sequence B (delete from A)
- **Horizontal:** Insert a gap in sequence A (delete from B)

The maximum of these three values represents the optimal score achievable for aligning $A[1..i]$ with $B[1..j]$. To facilitate traceback, implementations record which case yielded the maximum value.

PHASE 3: TRACEBACK Once the matrix is filled, $F(n, m)$ contains the optimal global alignment score. To reconstruct the actual alignment, traceback is performed: starting at $F(n, m)$, the algorithm works backwards to $F(0, 0)$ by determining which of the three possible transitions led to each cell's optimal score.

At each cell (i, j), the algorithm examines which predecessor contributed to the current score:

- **Diagonal predecessor** $F(i-1, j-1)$: Match/mismatch a_i with b_j, move to $(i-1, j-1)$
- **Vertical predecessor** $F(i-1, j)$: Gap in B, include a_i with gap in B', move to $(i-1, j)$
- **Horizontal predecessor** $F(i, j-1)$: Gap in A, include b_j with gap in A', move to $(i, j-1)$

Since the alignment is built from right to left, characters are prepended as the algorithm proceeds. Traceback terminates upon reaching $F(0, 0)$, yielding the complete optimal alignment.

When multiple predecessors yield the same maximum score, multiple optimal alignments exist with identical total scores. Common strategies include prioritizing diagonal moves, minimizing gaps, or choosing arbitrarily but consistently. A complete worked example of the Needleman-Wunsch algorithm applied to sequence alignment can be found in Appendix a.1.

2.5.3 Application to Trace Alignment

In the context of this thesis, Needleman-Wunsch is adapted to align NCCL operation sequences (from debug logs) with kernel execution sequences (from Nsys traces). The key adaptations include:

- **Alphabet:** Operation types (`AllReduce`, `Send`, `Recv`, etc.) rather than amino acids
- **Scoring:** Custom substitution matrix reflecting NCCL operation semantics and asymmetric gap penalties accounting for the different granularities of the two data sources
- **Constraints:** Exploiting temporal ordering and operation frequency to improve alignment accuracy

The detailed implementation and domain-specific adaptations are described in Section 4.2.2.

3 STATE OF THE ART AND RELATED WORK

Profiling distributed deep learning workloads requires understanding both high-level communication semantics, what data is being transferred, which communicator is involved, and low-level execution timing, when kernels execute, how long they take. However, existing profiling tools provide these views in isolation, making it difficult to analyze communication performance at the granularity needed for optimization. This chapter first establishes the methodology used to survey the literature, then examines the state of the art in distributed training profiling and analysis, reviews existing profiling tools and technologies, and finally identifies the specific research gap that this thesis addresses.

3.1 LITERATURE REVIEW METHODOLOGY

To establish the novelty of this work, a systematic literature search was conducted across major academic databases and search engines, including IEEE Xplore, the ACM Digital Library, and Google Scholar. The search focused on combinations of terms related to low-level communication analysis, specific parallelism strategies, and underlying communication libraries, using the search terms shown in Table 3.1 and Table 3.2.

Table 3.1: Search Queries for Establishing General Research Context

No.	Search Query	Rationale / Objective
1	`"NCCL" AND "distributed training" AND "performance"`	Identify existing NCCL performance analyses.
2	`("Megatron-LM" OR "DeepSpeed") AND "scalability"`	Review scalability studies of training frameworks.
3	`"tensor parallelism" AND "communication cost"`	Find analyses of communication overheads.
4	`"large language models" AND "training" AND "bottleneck"`	Survey common LLM training bottlenecks.

This systematic search revealed a significant body of work on high-level performance characterization and framework benchmarking.

Table 3.2: Search Queries to Identify the Specific Research Gap

No.	Search Query	Rationale / Objective
1	"NCCL" AND "network traffic" AND ("characterization" OR "analysis")	Search for direct NCCL traffic studies.
2	"tensor parallelism" AND "communication patterns" AND "visualization"	Look for work on visualizing communication.
3	("visualizing" OR "tracing") AND "NCCL" AND "communication"	Check for NCCL communication visualization tools.
4	("parsing" OR "analyzing") AND "NCCL logs" AND ("trace" OR "timeline")	Search for prior NCCL log-analysis methods.

Studies analyzing the scalability of Megatron-LM and DeepSpeed are well-represented in the literature, as are theoretical analyses of communication costs for various parallelism strategies. However, a critical gap emerges when examining work at the intersection of low-level network behavior and distributed training frameworks.

Existing research can be broadly categorized into three areas. First, framework-level performance studies measure end-to-end training throughput and demonstrate scalability improvements through novel parallelism algorithms, but these works treat network communication as an aggregate cost rather than examining fine-grained traffic patterns. Second, theoretical analyses provide mathematical models of communication complexity for different parallelism strategies, offering valuable insights into asymptotic behavior but lacking empirical validation of real-world traffic characteristics. Third, a limited number of studies examine communication overhead in isolation using synthetic benchmarks.

The following sections examine these research areas in detail, followed by a review of existing profiling tools and a synthesis identifying the specific research gap.

3.2 STATE OF THE ART IN DISTRIBUTED TRAINING PROFILING

This section surveys research on distributed training performance, examining what kinds of profiling studies have been conducted and what insights they have revealed about communication behavior in large-scale training workloads.

3.2.1 Characterization and Performance Modeling Studies

Several recent studies have characterized the performance of distributed training workloads through empirical profiling and modeling. Go et al. [31] conducted a comprehensive efficiency characterization of large language model training, analyzing how different parallelization strategies affect GPU utilization, memory consumption, and communication overhead across various cluster configurations. Their work reveals that communication can consume 40-60% of wall-clock time depending on the parallelization configuration, highlighting the critical importance of understanding communication behavior.

Hanindhito, Patel, and John [34] performed detailed bandwidth characterization of multi-GPU systems, measuring achievable throughput for different communication patterns and network topologies. Their results demonstrate significant variance in achieved bandwidth depending on message size, communicator topology, and concurrent operation patterns, variations that are difficult to observe without fine-grained profiling.

Performance modeling approaches have attempted to predict training time and resource requirements before deployment. Liang et al. [49] introduced Lumos, a performance modeling framework for large-scale LLM training that estimates end-to-end training time by modeling computation, communication, and memory access patterns. Kundu et al. [45] developed analytical models for predicting training performance across different hardware configurations. While these modeling approaches provide valuable insights, they rely on either coarse-grained measurements or simplifying assumptions about communication behavior, limiting their accuracy for detecting subtle performance issues or validating actual communication patterns against predictions.

The Calculon framework [40] provides a methodology for co-designing systems and large language models through simulation, enabling exploration of hardware-software trade-offs before physical deployment. However, simulation-based approaches necessarily abstract away implementation details and may not capture emergent behaviors that occur in real deployments, such as network congestion, operating system scheduling effects, or subtle interactions between overlapped operations.

3.2.2 Communication Analysis in Production Systems

Industry-scale deployments have provided valuable insights into communication challenges at extreme scales. Jiang et al. [42] describe ByteDance's MegaScale system for training models with up to 175 billion parameters across thousands of GPUs, documenting the communication optimizations and failure-handling mechanisms required

for production training. Their work emphasizes that communication reliability and observability become critical at scale, yet existing profiling tools provide limited visibility into the actual behavior of communication libraries during training.

Singhania et al. [73] analyze communication bottlenecks in multi-node LLM inference, identifying that `AllReduce` operations for TP frequently become the limiting factor for throughput. Wang et al. [83] propose hiding communication costs through micro-batch co-execution, demonstrating 15-20% training speedups by carefully overlapping computation and communication. Li et al. [47] introduce Flash Communication to reduce TP bottlenecks through optimized communication scheduling.

While these works provide important insights into communication optimization strategies, they typically measure end-to-end training time or aggregate communication statistics rather than analyzing individual collective operations. Understanding why a particular optimization works, or which specific operations are causing bottlenecks, requires finer-grained visibility into the relationship between NCCL's algorithm selection, message sizes, and actual kernel execution timing.

3.2.3 Framework-Specific Profiling and Debugging

Recognizing the gap between generic profiling tools and the needs of distributed training developers, researchers have created framework-specific analysis tools.

Zhao et al. [87] introduced MegatronApp, a tracing framework designed specifically for Megatron-LM that instruments the training loop to capture communication patterns, identify slow nodes, and detect stragglers in distributed training. MegatronApp provides valuable insights into node-level performance variations and can identify when specific ranks are slower than others. However, it still treats NCCL communication at a coarse granularity, without reporting details such as which algorithm was selected or what message size was used.

Similarly, production training frameworks like DeepSpeed [66] and Megatron-LM include built-in monitoring capabilities that track high-level metrics like throughput, memory usage, and communication time. These tools are invaluable for production monitoring and detecting gross inefficiencies, but they lack the granularity needed to answer detailed questions about individual communication operations or to understand the relationship between parallelization decisions and network utilization patterns.

3.2.4 Communication Overhead Quantification

Multiple studies have quantified the impact of communication overhead on training performance. Ouyang et al. [58] provide a compre-

hensive survey of communication optimization strategies, categorizing approaches into gradient compression, communication scheduling, and topology-aware algorithms. Hassan [35] further analyze how the proportion of time spent in communication varies with model architecture, parallelization strategy, and network topology.

These quantitative studies establish that communication is a critical performance factor, yet the measurements are typically performed at a coarse granularity (total time in communication vs. computation) or focus on aggregate metrics (average bandwidth, total data transferred). Fine-grained analysis of individual collective operations, understanding which specific `AllReduce` or `AllGather` operations dominate communication time, why NCCL selected a particular algorithm, requires tooling that can correlate NCCL's semantic metadata with precise kernel timing.

3.3 EXISTING PROFILING TOOLS AND TECHNOLOGIES

Having established what kinds of profiling studies exist in the literature, this section examines the tools themselves that enable these studies, and more importantly, their limitations that prevent more detailed analysis.

3.3.1 Framework-Level Profilers

Framework-integrated profilers operate at the level of deep learning operations, providing visibility into the logical structure of training workloads. PyTorch Profiler [64] and TensorFlow Profiler [51] capture high-level operator names, tensor shapes, and Python call stacks, enabling identification of algorithmic bottlenecks within the training loop.

However, framework-level profilers face fundamental limitations when analyzing distributed training [67]. Collective communication operations appear as opaque library calls without visibility into the actual data movement patterns.

3.3.2 Hardware-Level Profilers

Hardware-level profiling tools provide detailed visibility into GPU execution without framework semantics. NVIDIA Nsight Systems (Nsys) [19] offers timeline visualization with nanosecond precision, capturing GPU kernel launches, CUDA API calls, and memory transfers. Nsight Compute provides deeper analysis of individual kernel performance, including instruction-level metrics and memory bandwidth utilization.

Nsys captures the actual NCCL kernel executions on the GPU timeline, including precise start and end times for each communication operation. While NCCL internally instruments its operations with NVIDIA Tools Extension (NVTX) annotations containing this metadata in the raw `.nsys-rep` trace format, these annotations are deliberately excluded during the SQLite export process, making them inaccessible to standard post-processing workflows [10]. Consequently, collective operations appear as anonymous CUDA kernel invocations without information about what data they transfer or which logical communicator they belong to. This makes it impossible to answer basic questions like *"How much bandwidth did this `AllReduce` achieve?"* or *"Why is this specific operation slow?"*.

3.3.3 Communication Library Profiling

NCCL itself provides built-in debug logging through environment variables (`NCCL_DEBUG=INFO`) [16]. These logs contain rich semantic information: message sizes, data types, and communicator IDs. However, the debug logs are textual outputs that lack precise timing information, they record when operations are enqueued but not when they actually execute on the GPU or how long they take.

Recent work has provided deeper understanding of NCCL's internal mechanisms. Hu et al. [38] conducted a comprehensive analysis of NCCL's algorithms, bandwidth utilization, and protocol overhead, revealing performance characteristics that are otherwise opaque to users. Production deployments have identified additional challenges: Chen et al. [4] document limitations in peer-to-peer efficiency and observability at scales exceeding 100,000 GPUs, while Si et al. [72] describe Meta's NCCLX framework addressing similar issues. These studies demonstrate that understanding NCCL's behavior is critical for optimization, yet the standard profiling workflow provides no way to connect NCCL's semantic metadata with actual kernel execution timing.

Recognizing these limitations, recent tools have emerged to provide more specialized communication profiling capabilities. Vardas, Rodriguez, and Beni [81] introduced ncclsee, a lightweight profiler plugin that leverages NCCL's version 2 profiling interface together with the CUDA Profiling Tools Interface (CUPTI) to capture communication patterns in real time. Unlike tools such as Nsys that produce highly detailed but unwieldy traces, ncclsee provides summary information aggregated by operation type and buffer size range, allowing users to quickly identify communication patterns and potential bottlenecks. A key technical contribution of ncclsee is its approach to correlating NCCL's asynchronous profiling events with CUPTI kernel timing, addressing the challenge that NCCL's `stopEvent` callback indicates only that a collective has been enqueued rather than completed. In-

tegration requires only setting an environment variable to load the profiler plugin, making it applicable to any application using NCCL directly or through frameworks like PyTorch.

Issa et al. [41] developed Snoopie, an instrumentation-based multi-GPU communication profiler built on NVBit that takes a complementary approach to communication analysis. While ncclsee focuses on NCCL collective operations, Snoopie provides broader coverage of GPU-to-GPU data movement including peer-to-peer transfers and direct memory accesses. Snoopie can attribute data movement to source code lines and data objects, offering visualization modes at varying granularities from system-wide overviews to specific instructions and addresses. The tool supports both NCCL and NVSHMEM communication libraries, and includes an interactive visualization module for exploring communication patterns. However, Snoopie's instrumentation-based approach introduces runtime overhead that may be prohibitive for production training workloads, and its focus on data movement attribution differs from the semantic analysis of collective operation parameters that practitioners need for debugging algorithm selection and bandwidth issues.

3.4 THE RESEARCH GAP

The systematic review reveals that while communication is widely recognized as a critical performance factor, existing tools and methodologies remain fundamentally limited in their ability to analyze communication behavior at sufficient granularity.

Recent tools such as ncclsee and Snoopie represent important advances in communication profiling. Ncclsee provides aggregated statistics by operation type and buffer size range, offering a practical summary view of NCCL behavior without the overwhelming volume of detailed traces. Snoopie enables attribution of data movement to source code and data objects, with support for multiple visualization granularities. However, both tools address different aspects of the profiling problem than the one this thesis targets.

Ncclsee's aggregation approach, while useful for identifying broad patterns, loses the per-operation detail needed to correlate specific collectives with their execution context, practitioners cannot determine which individual `AllReduce` within a training iteration achieved poor bandwidth or why a particular operation selected an unexpected algorithm. Snoopie's instrumentation-based approach provides fine-grained data movement tracking but focuses on memory access patterns rather than the semantic parameters (algorithm selection, communicator topology, message size) that determine NCCL's performance characteristics. Neither tool provides a mechanism to correlate

NCCL's rich debug metadata with precise kernel timing on a per-operation basis.

Critically, no identified work empirically characterizes the fine-grained communication patterns that emerge during actual LLM training with different parallelism strategies. While the theoretical communication costs of the parallelism types are well understood as outlined in Section 2.4, the absence of tools that allow systematic analysis of real-world NCCL traffic at operation-level granularity means that practitioners lack visibility into the actual network behavior of their training runs.

The following questions remain unanswerable with existing tools:

- **Communication structure analysis:** What are the precise operation type distributions, message size characteristics, and temporal patterns for each parallelism strategy? How do these signatures differ across configurations?
- **Theoretical volume validation:** Do actual communication volumes match theoretical predictions for a given parallelization strategy? Where and why do observed volumes deviate from idealized formulas?
- **Per-operation characterization:** Which specific operations dominate communication time? How are message sizes distributed across operation types, and how does this vary with parallelism configuration?

The core challenge is that the information needed to answer these questions exists but is split across two incompatible data sources: NCCL's debug logs contain rich semantic metadata (message sizes, algorithms, communicator IDs), while Nsys traces contain precise kernel timing. While tools like ncclsee and Snoopie provide valuable capabilities for communication profiling, neither bridges this specific gap, ncclsee aggregates rather than correlates individual operations, and Snoopie focuses on memory access patterns rather than NCCL semantics. This leaves researchers unable to characterise communication structure at the operation level, validate theoretical volume models against implementation reality, or understand how parallelism configurations manifest as distinct network traffic signatures.

This thesis addresses the identified gap by developing a methodology to correlate NCCL debug logs with Nsys kernel traces, enabling fine-grained, per-operation analysis of communication performance during distributed training. The approach works with standard profiling outputs, `NCCL_DEBUG=INFO` logs and `nsys export` SQLite databases, requiring no source code modifications, custom instrumentation, or specialized hardware support. By treating the correlation problem as a sequence alignment task, the methodology establishes correspondences between semantic metadata and timing information despite

temporal desynchronization, granularity mismatches, and the absence of common identifiers. The following chapter describes the technical design and implementation of this trace correlation methodology.

4 NCCL TRACE PROFILER

The previous chapter established that existing profiling tools provide either semantic metadata or precise timing, but not both, leaving practitioners unable to perform fine-grained analysis of communication performance. This chapter presents the NCCL Trace Profiler, a Python-based tool that addresses this gap by correlating NCCL debug logs with Nsys kernel traces using sequence alignment techniques.

The profiler works out-of-the-box with standard profiling outputs, `NCCL_DEBUG=INFO` logs and Nsys SQLite database exports, requiring no source code modifications or custom instrumentation. By implementing the Needleman-Wunsch sequence alignment algorithm, it maps logical NCCL operations to physical CUDA kernels despite temporal desynchronization, granularity mismatches, and the absence of common identifiers.

Optional features include NVTX annotation support for iteration-specific analysis and clock synchronization for multi-node temporal alignment. All results are exported to the Chrome Trace Event Format, enabling interactive visualization in standard trace viewers.

This chapter is structured as follows: Section 4.1 formally defines the technical problem and solution approach, Section 4.2 provides a high-level architecture overview, and subsequent sections detail the implementation of data ingestion, sequence alignment, communicator inference, metric calculation, and visualization.

4.1 THE TRACE CORRELATION CHALLENGE

Having established the practical need for fine-grained communication profiling, this section formally defines the technical challenge of correlating NCCL debug logs with Nsys kernel traces. Understanding this challenge is essential for appreciating the design decisions in the subsequent Section 4.2.

4.1.1 Why Direct Correlation Fails

The information needed to analyze communication performance is split across two data sources with fundamentally different characteristics. These two data sources cannot be directly joined due to fun-

damental mismatches between how operations are logged and how they execute on hardware. Empirical analysis of production training workloads reveals the complexity of this correlation challenge.

OPERATION FUSION AND DECOMPOSITION NCCL internally fuses or decomposes operations in ways that make one-to-one matching impossible. Table 4.1 shows operation counts from a TP=2, PP=2 training run on a 4-GPU node, revealing systematic mismatches between logged operations and executed kernels.

Table 4.1: Operation Count Mismatch: TP=2, PP=2 Configuration (4 GPUs)

Rank	Operation	Log Count	Nsys Count	Delta
1	AllGather	42	21	+21
	AllReduce	5942	2966	+2976
	Broadcast	1922	961	+961
	Recv	320	0	+320
	Send	320	0	+320
	SendRecv	0	330	-330
2	AllGather	21	21	0
	AllReduce	2987	2966	+21
	Broadcast	961	961	0
	Recv	320	0	+320
	Send	320	0	+320
	SendRecv	0	330	-330
3	AllGather	21	21	0
	AllReduce	7786	3926	+3860
	Broadcast	1921	961	+960
	Recv	320	0	+320
	Send	320	0	+320
	SendRecv	0	330	-330
4	AllGather	21	21	0
	AllReduce	3946	3926	+20
	Broadcast	961	961	0
	Recv	320	0	+320
	Send	320	0	+320
	SendRecv	0	330	-330

The most striking pattern is the `Send`/`Recv` fusion: NCCL logs record 320 `Send` and 320 `Recv` operations per rank, but the hardware executes 330 fused `SendRecv` kernels. This represents an internal optimization where bidirectional point-to-point transfers are combined into a single kernel invocation for efficiency. However, the correspondence is

not merely fusion. There are 10 additional `SendRecv` kernels executed beyond what would result from simply pairing the 320 logged `Send` operations with their corresponding `Recv` operations. This +10 discrepancy suggests that some `SendRecv` kernels are executed without corresponding entries in the debug logs, possibly due to internal NCCL communication for coordination or metadata exchange that bypasses the standard logging path. Any correlation approach relying on exact operation name matching or even simple fusion rules would fail immediately on this pattern alone.

RANK-DEPENDENT KERNEL BATCHING Beyond operation fusion, the traces reveal rank-dependent batching behavior that varies with the TP configuration. Table 4.2 shows operation counts from a TP=4 configuration, where one rank exhibits systematic 2:1 ratios between logged operations and executed kernels, while the remaining three ranks show nearly perfect correspondence.

Table 4.2: Operation Count Mismatch: TP=4 Configuration (4 GPUs)

Rank	Operation	Log Count	Nsys Count	Delta
1	AllGather	8682	4341	+4341
	AllReduce	1470	725	+745
	Broadcast	1602	801	+801
	ReduceScatter	5440	2720	+2720
2	AllGather	4341	4341	0
	AllReduce	745	725	+20
	Broadcast	801	801	0
	ReduceScatter	2720	2720	0
3	AllGather	4341	4341	0
	AllReduce	745	725	+20
	Broadcast	801	801	0
	ReduceScatter	2720	2720	0
4	AllGather	4341	4341	0
	AllReduce	745	725	+20
	Broadcast	801	801	0
	ReduceScatter	2720	2720	0

Rank 1 consistently shows exactly double the logged operations compared to executed kernels across all operation types: 8682 logged `AllGather` operations result in only 4341 kernels, 1470 `AllReduce` operations produce 725 kernels, and so forth.

The TP=2, PP=2 configuration in Table 4.1 reveals a similar but more widespread pattern. Ranks 1 and 3 both exhibit 2:1 ratios for collective operations: rank 1 shows 42 logged `AllGather` operations

producing 21 kernels and 5942 `AllReduce` operations producing 2966 kernels, while rank 3 shows 7786 `AllReduce` operations producing 3926 kernels and 1921 `Broadcast` operations producing 961 kernels. Meanwhile, ranks 2 and 4 show nearly perfect 1:1 correspondence for most operations, with only a persistent +20 `AllReduce` discrepancy that is discussed below.

This batching behavior follows a predictable pattern based on TP topology. In the TP=4 configuration on 4 GPUs, exactly $\frac{4}{4} = 1$ rank exhibits 2:1 batching. In the TP=2 configuration, $\frac{4}{2} = 2$ ranks show 2:1 ratios. The number of ranks exhibiting batching behavior equals the number of tensor-parallel groups in the cluster.

The source of this batching is not immediately apparent from the traces. Since NCCL logs record operations at the point they are enqueued to the GPU, the 2:1 ratio must arise from kernel-level optimizations, either within NCCL's GPU kernel implementations, the CUDA runtime's kernel fusion capabilities, or low-level driver optimizations that coalesce operations after they leave the NCCL logging layer. Whether this batching is an intentional NCCL design decision, an emergent property of GPU scheduling under Megatron-LM's communication patterns, or an interaction between framework and library remains unclear without deeper instrumentation. What is certain is that this topology-dependent behavior cannot be predicted a priori and demonstrates why generic correlation approaches that assume uniform operation-to-kernel mappings across all ranks will produce incorrect results.

THE PERSISTENT +20 ALLREDUCE ANOMALY Across both configurations, every rank exhibits a consistent pattern: there are approximately 20 more `AllReduce` operations in the logs than in the kernel traces. In Table 4.1, ranks show +20, +21, +20, and +20 excess logged operations. In Table 4.2, all four ranks exhibit exactly +20. This systematic discrepancy suggests that certain `AllReduce` operations logged by NCCL are either optimized away at the kernel level, handled through alternative mechanisms, or represent operations where trace collection terminated before kernel execution completed.

4.1.2 Sequence Alignment as the Solution Approach

The key insight enabling correlation is that despite these mismatches, NCCL debug logs and Nsys kernel traces share a structural similarity: both represent sequences of collective operations that follow the same underlying execution order imposed by the training framework.

The Needleman-Wunsch sequence alignment algorithm, described in Section 2.5, provides a solution to this correlation problem. The algorithm can establish correspondences despite temporal desynchronization and granularity differences. Crucially, the algorithm allows

insertions and deletions to handle the many-to-many mappings observed in Tables 4.1 and 4.2.

4.2 PROFILER ARCHITECTURE

The NCCL Trace Profiler follows a staged pipeline architecture that transforms standard profiling artifacts into enriched communication traces. As illustrated in Figure 4.1, the profiler operates in three conceptual phases: data ingestion, sequence alignment, and trace enrichment and generation.

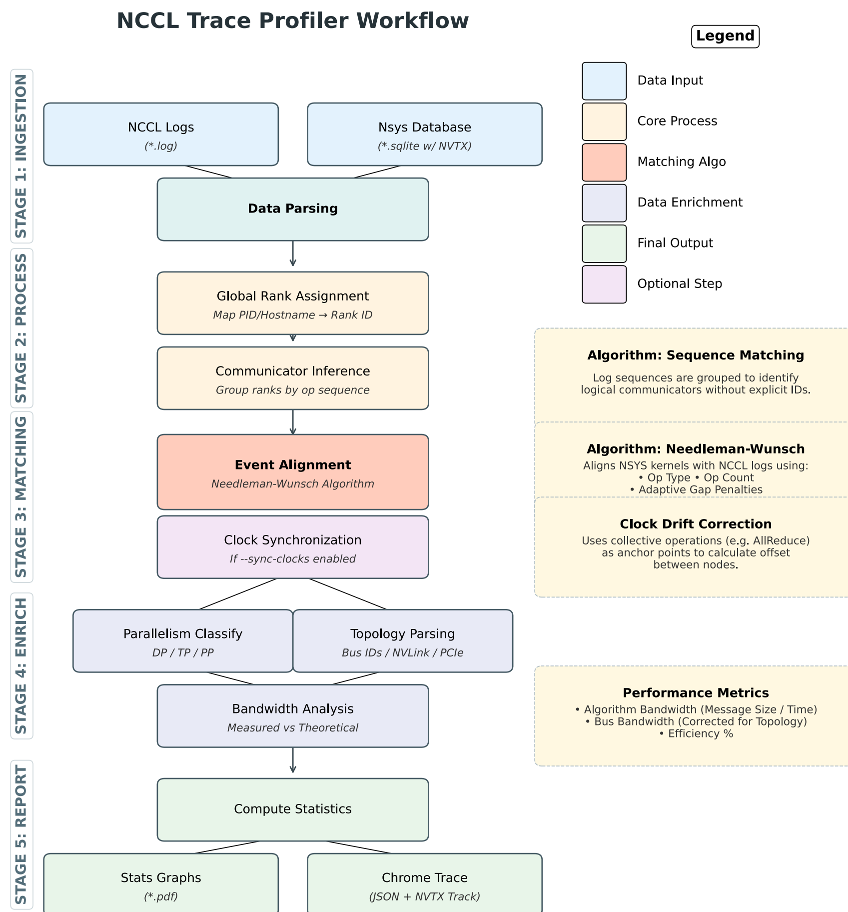


Figure 4.1: End-to-end workflow of the NCCL Trace Profiler, illustrating the staged pipeline from data ingestion through sequence alignment, enrichment, and trace generation.

In the ingestion phase, the profiler parses NCCL debug logs and Nsys SQLite databases and establishes a consistent global view of ranks, communicators, and execution context across data sources. The alignment phase correlates communication events recorded in logs with GPU kernel executions using a sequence alignment algorithm,

enabling precise temporal matching despite asynchronous execution and instrumentation noise. Finally, the enrichment and output phase augments aligned operations with semantic metadata and performance metrics, producing Chrome Trace Format output and summary statistics for analysis.

Each stage is described in detail in the following sections.

4.2.1 Data Ingestion and Parsing

The profiling pipeline ingests and fuses data from two primary sources to reconstruct the communication lifecycle: NCCL debug logs for semantic metadata and Nsys SQLite exports for timing information.

NCCL DEBUG LOG PARSING Generated via the environment variable `NCCL_DEBUG=INFO`, these logs provide logical metadata of operations within the CPU space. While they do not contain high-level framework names, they provide critical bridge data: the operation type (`AllReduce`), the `opCount`, the specific `comm` pointer, and the chosen `Algo` (e.g., `RING`).

The profiler implements a custom regex-based parser to extract structured information from these logs:

```
1766081276.802766 csg-rivulet02:1426907:1427588 [2] NCCL INFO AllReduce:
    opCount 0 sendbuff 0x7fd9f7780000 recvbuff 0x7fd9f7780000 count
    131072 datatype 6 op 0 root 0 comm 0x447b8890 stream 0x32dc6760
1766081276.802821 csg-rivulet02:1426907:1427588 [2] NCCL INFO AllReduce:
    262144 Bytes -> Algo RING proto LL channel{Lo..Hi}={0..7}
```

Listing 4.1: Example NCCL Debug Output

Each line identifies:

- **Timestamp:** Epoch-based CPU time in seconds from the operating system's monotonic clock.
- **Hostname and PID/TID:** `csg-rivulet02:1426907:1427588` uniquely identifies the process and thread emitting the log.
- **Local Rank:** `[2]` identifies GPU 2 on this node (0-indexed).
- **Operation:** `AllReduce`, `AllGather`, `ReduceScatter`, `Broadcast`, `Send`, `Recv`, or `SendRecv`.
- **Parameters:** Buffer pointers (`sendbuff`, `recvbuff`), element count (131072), data type enumeration (6 = float16).
- **Communicator:** `0x447b8890` uniquely identifies the logical group within this process.
- **Algorithm:** `RING` indicates the selected communication pattern. Other algorithms include `TREE`, `COLLNET_DIRECT`, and `COLLNET_CHAIN`.

- **Protocol:** `LL` (Low Latency), `LL128`, or `SIMPLE`, which affects bandwidth efficiency.
- **Channels:** `channel{Lo..Hi}={0..7}` indicates which NCCL channels are used for this operation.

Because NCCL's debug log format has remained stable across major releases, this parsing approach is robust to version changes and does not require maintenance when upgrading NCCL.

The parser converts NCCL data type enumerations [15] to human-readable names:

Table 4.3: NCCL Data Type Enumeration Mapping

Enum	Type
0	int8
1	uint8
6	float16
7	float32
8	float64
9	bfloat16

Similarly, reduction operation enumerations are decoded:

Table 4.4: NCCL Reduction Operation Enumeration Mapping

Enum	Operation
0	sum
1	prod
2	max
3	min
4	avg

The parser also extracts communicator size information from initialization lines:

```
1 ncclCommInitRankConfig comm 0x447b8890 rank 2 nranks 4 ...
```

Listing 4.2: Communicator Initialization

This provides the mapping from communicator pointer to the number of participating ranks, essential for bandwidth calculations.

TOPOLOGY DISCOVERY AND ANALYSIS Topology data is extracted directly from the NCCL debug logs. NCCL emits detailed topology information during initialization that captures the runtime view of interconnects, including measured bandwidth values and the actual communication paths that NCCL will use. This approach provides

accurate implementation-specific topology information, as it reflects NCCL's internal topology model.

The topology parser implements a state machine to extract hierarchical connectivity information:

1. **System-Level Parsing:** The parser first identifies the system header block marked by `=== System : maxBw X.X totalBw Y.Y ===`, which provides aggregate bandwidth metrics.

2. **Bus ID Extraction:** Each GPU entry contains a unique bus ID (e.g., `0-1000`, `0-c1000`) that serves as a persistent identifier across profiling runs. The parser extracts these IDs using the pattern `GPU/0-<bus_id> (<local_rank>)`, building a bidirectional mapping between bus IDs and local ranks.

3. **Link Topology Construction:** For each GPU, the parser traverses indented child entries to identify:
   - **PCI links:** `PCI[X.X]` indicates bandwidth to CPU socket
   - **NVLink connections:** `NVL[X.X] - GPU/0-<target_bus_id>` indicates GPU-to-GPU high-speed links
   - **NVSwitch connections:** `NVL[X.X] - NVS/0-0` indicates switch-based topology
   - **Network interfaces:** `NET[X.X] - NET/0-N` indicates RDMA-capable NICs

4. **Multi-Block Handling:** NCCL logs may contain multiple topology blocks (one per communicator initialization). The parser targets the first block, as subsequent blocks may contain partial or redundant information for sub-communicators.

This topology information is subsequently used for determining communication bottlenecks and calculating theoretical bandwidth limits.

Listing 4.3 shows a representative NCCL topology excerpt that captures connectivity between CPUs, GPUs, and network interfaces, along with measured link bandwidths.

```
NCCL INFO === System : maxBw 80.0 totalBw 80.0 ===

NCCL INFO CPU/0-0 (1/2/-1)
NCCL INFO + PCI[24.0] - GPU/0-1000 (0)
NCCL INFO                 + NVL[80.0] - GPU/0-25000
NCCL INFO + PCI[24.0] - GPU/0-25000 (1)
NCCL INFO                 + NVL[80.0] - GPU/0-1000
NCCL INFO + SYS[16.0] - CPU/0-1

NCCL INFO CPU/0-1 (1/2/-1)
NCCL INFO + PCI[24.0] - GPU/0-c1000 (2)
NCCL INFO                 + NVL[80.0] - GPU/0-e1000
NCCL INFO + PCI[24.0] - GPU/0-e1000 (3)
```

```
14 NCCL INFO              + NVL[80.0] - GPU/0-c1000
15 NCCL INFO + SYS[16.0] - CPU/0-0
16 NCCL INFO + PCI[12.0] - NIC/0-c2000
17 NCCL INFO              + NET[12.5] - NET/0-0
18 NCCL INFO + PCI[12.0] - NIC/0-c3000
19 NCCL INFO              + NET[12.5] - NET/0-1
```

Listing 4.3: Example NCCL topology graph extracted during initialization.

Each CPU socket is represented as a root node (e.g., CPU/0-0 and CPU/0-1), reflecting a dual-socket Non-Uniform Memory Access (NUMA) configuration. GPUs attached via `PCI` links are shown as children of the corresponding CPU, with bracketed values indicating measured link bandwidth in gigabytes per second and unique bus IDs (e.g., `0-1000`, `0-25000`) that identify each GPU. GPU pairs connected via NVLink are denoted by `NVL` edges with significantly higher bandwidth. The presence of `SYS` links between CPU nodes indicates cross-NUMA communication paths. Network Interface Controllers (NICs) are similarly attached via PCIe and connected to logical network endpoints via `NET` links.

The profiler parses this topology structure to build a complete interconnect map using bus IDs as unique identifiers. This bus ID-based approach is robust across different nodes and profiling runs, as NCCL consistently uses these identifiers.

NSYS SQLITE PARSING The `nsys export` command produces SQLite databases containing kernel execution traces. The profiler queries the `CUPTI_ACTIVITY_KIND_KERNEL` table joined with `StringIds` to extract demangled kernel names
(e.g., `ncclDevKernel_AllReduce_RING_LL_Sum_int8`), start and end timestamps in the GPU clock domain, process ID (`pid`) for correlating with NCCL logs, kernel launch configuration (grid dimensions, block dimensions, shared memory usage), and stream ID and correlation ID.

The SQL query structure is:

```
1  SELECT
2      p.pid, Name.value AS demangledName,
3      Kernel.start, Kernel.end,
4      Kernel.gridX, Kernel.gridY, Kernel.gridZ,
5      Kernel.blockX, Kernel.blockY, Kernel.blockZ,
6      Kernel.staticSharedMemory, Kernel.dynamicSharedMemory,
7      Kernel.registersPerThread,
8      Kernel.streamId, Kernel.correlationId
9  FROM CUPTI_ACTIVITY_KIND_KERNEL AS Kernel
10 JOIN StringIds AS Name ON Kernel.demangledName = Name.id
11 JOIN PROCESSES AS p ON Kernel.globalPid = p.globalPid
12 WHERE Name.value LIKE '%nccl%'
13 ORDER BY Kernel.start
```

Listing 4.4: Nsys SQLite Query

The kernel names are parsed using regular expressions to extract the operation type and algorithm.
The pattern `nccl(?:Dev)?Kernel_([a-zA-Z0-9]+)[_\(]` matches kernel names like `ncclDevKernel_AllReduce_Sum_f16_RING_LL` and `ncclDevKernel_SendRecv`. For ambiguous or malformed names, the profiler falls back to substring matching (`"AllReduce" in raw_name`).

The profiler can process multiple SQLite files simultaneously (one per node), building a per-process event list indexed by PID. This enables efficient parallel processing and subsequent PID-based matching with NCCL logs.

NVTX ANNOTATION EXTRACTION (OPTIONAL) When applications instrument their code with NVTX markers, the profiler can extract these annotations from the `NVTX_EVENTS` table in the SQLite database. These annotations provide semantic boundaries for training iterations, forward/backward passes, pipeline stages, and other application-specific phases. The NVTX extractor module processes these annotations and creates a separate track in the Chrome trace visualization showing iteration boundaries and training phases.

This functionality is entirely optional. The profiler operates without any NVTX data, but when present, it enables filtering analysis to specific training iterations, correlating communication patterns with training phases, visualizing iteration boundaries alongside NCCL operations, and identifying iteration-to-iteration variations in communication.

The NVTX extractor implements deduplication logic to prevent showing redundant markers across multiple GPU threads, selecting a representative thread with the most complete hierarchy of annotations.

4.2.2 Sequence Alignment via Needleman-Wunsch

The Needleman-Wunsch algorithm [55], described in detail in Section 2.5, is adapted here to align sequences of NCCL operation names.

KEY ADAPTATIONS FOR TRACE ALIGNMENT: The application of Needleman-Wunsch to trace alignment differs from its traditional bioinformatics usage in several critical ways:

1. **Alignment Domain:** Instead of matching characters from a 4-letter (DNA) or 20-letter (protein) alphabet, it matches operation names from the set {`AllReduce`, `AllGather`, `ReduceScatter`, `Broadcast`, `Send`, `Recv`, `SendRecv`}.

2. **Asymmetric Penalty Structure:** Traditional sequence alignment uses symmetric penalties (mismatch $\approx$ gap). For trace alignment, it employs a harsh asymmetric penalty structure:
   - *Match score:* $+5$ (weighted by operation importance)

- *Mismatch penalty:* −15 (3× larger than gap penalty)
- *Gap penalty:* −5 (adaptive, increasing with consecutive gaps)

This asymmetry is critical: it strongly discourages aligning operations of different types, e.g., matching an `AllReduce` kernel with a `Broadcast` log entry, instead preferring to introduce gaps. This prevents false matches that would corrupt bandwidth calculations.

3. **Operation-Specific Weighting:** Different operation types receive different weights to reflect their relative prevalence and alignment confidence:

$$w(op) = \begin{cases} 1.0 & \text{if } op = \texttt{AllReduce} \text{ (baseline)} \\ 2.0 & \text{if } op \in \{\texttt{ReduceScatter}, \texttt{AllGather}, \texttt{Broadcast}\} \\ 0.5 & \text{if } op \in \{\texttt{Send}, \texttt{Recv}, \texttt{SendRecv}\} \end{cases} \tag{4.1}$$

Rare operations receive higher weights to increase their influence on alignment decisions, as matching them provides stronger evidence of correct correlation.

4. **Adaptive Gap Penalties:** Consecutive gaps incur progressively higher penalties to discourage long runs of unmatched operations:

$$g(k) = -5 \times (1 + r_k \times 0.3) \tag{4.2}$$

where r_k is the length of the current gap run. This reflects the domain knowledge that distributed training operations typically execute in consistent patterns, making long gap sequences indicative of misalignment.

SCORING FUNCTION The modified scoring function for matching operations n_i from Nsys and l_j from NCCL logs is:

$$s(n_i, l_j) = \begin{cases} 5 \times w(n_i) & \text{if } n_i = l_j \text{ (match)} \\ -15 \times \frac{w(n_i)+w(l_j)}{2} & \text{if } n_i \neq l_j \text{ (mismatch)} \end{cases} \tag{4.3}$$

The alignment parameters (match score, mismatch penalty, gap penalty, operation weights) were determined through systematic hyperparameter search, using the validation suite detailed in Section 4.3. The current parameter set (MATCH = 5, MISMATCH = −15, GAP = −5) achieves the best balance between avoiding false positives and capturing true matches across diverse workloads.

METADATA FUSION For each matched pair, the analyzer copies metadata from the NCCL log to the Nsys kernel:

- Element count and data type for message size calculation
- Communicator pointer for logical group membership matching
- Algorithm selection for bandwidth correction factor
- Operation count as a unique identifier for debugging

Unmatched kernels retain their timing information but lack semantic metadata, flagged for manual inspection.

4.2.3 Communicator Inference and Global Rank Assignment

NCCL communicators are identified by memory pointers (e.g., `0x447b8890`), which differ across processes even for the same logical group. The analyzer infers logical communicator identities by analyzing operation sequences: communicators that execute the same sequence of operations across different ranks are considered part of the same logical group.

SEQUENCE-BASED MATCHING For each communicator pointer observed in the logs, the analyzer extracts the sequence of operation names:

$$S_c = \langle \mathrm{op}_1, \mathrm{op}_2, \ldots, \mathrm{op}_n \rangle \tag{4.4}$$

Communicators across ranks with identical sequences are grouped together and assigned a logical identifier:

$$\mathrm{LogicalComm}_k = \{ c \in \mathcal{C} \mid S_c = S_k \} \tag{4.5}$$

GLOBAL RANK CALCULATION Each process has a local rank (0-7 for an 8-GPU node) and a hostname. The analyzer assigns global ranks using:

$$\mathrm{GlobalRank} = (\mathrm{NodeIndex} \times \mathrm{GPUsPerNode}) + \mathrm{LocalRank} \tag{4.6}$$

where `NodeIndex` is determined by sorting hostnames alphabetically. This ensures consistent rank numbering across profiling runs.

COMMUNICATOR CLASSIFICATION Based on communicator size and participating ranks, the analyzer classifies each collective operation's parallelism type. Point-to-point operations are classified as PP, as these are typically used for inter-stage communication in pipelined models. For collective operations, user-specified parallelism sizes are used when available. When explicit parallelism sizes are not fully specified, the analyzer falls back to an inference strategy for DP. In this case,

communicators with the largest number of participating ranks are assumed to correspond to data-parallel collectives, provided that DP is active. Operations that cannot be classified under these rules are labeled generically as collective communication or unknown when insufficient information is available.

4.2.4 Clock Synchronization

Multi-node profiling introduces clock skew: each node's timestamps originate from independent clocks with potentially different baselines. To enable temporal alignment across nodes, the analyzer implements an optional clock synchronization mechanism that leverages collective operations as natural synchronization points.

SYNCHRONIZATION PRINCIPLE Collective operations (e.g., `AllReduce`) require barrier-like coordination: all participating ranks must contribute data before any can obtain the final result, though ranks may begin executing the collective kernel before all peers have entered the operation. While not strict barriers, the analyzer assumes that kernels with the same logical operation count across ranks represent the same collective instance and therefore should have temporally aligned midpoints.

OFFSET CALCULATION For each valid collective operation:

1. Extract kernel start times: t_r for each rank r
2. Compute median timestamp: $t_{\text{ref}} = \text{median}(\{t_0, t_1, \ldots, t_{N-1}\})$
3. Calculate raw offsets: $\delta_r = t_{\text{ref}} - t_r$
4. Collect offsets across multiple collectives
5. Take median offset per rank

REFERENCE RANK NORMALIZATION Choose rank 0 as the reference:

$$\delta_r^{\text{final}} = \delta_r - \delta_0 \tag{4.7}$$

This ensures rank 0 has zero offset, simplifying interpretation.

APPLICATION For each event with global rank r:

$$t_{\text{synchronized}} = t_{\text{original}} + \delta_r^{\text{final}} \tag{4.8}$$

Events retain both original and synchronized timestamps for verification.

LIMITATIONS The approach assumes relatively low jitter in the entry times of collective operations across ranks, such that temporal alignment remains stable. It further requires a sufficient number of repeated collective operations, typically more than ten, to enable robust median-based estimation. Finally, the method does not explicitly correct for clock drift between ranks and instead assumes a constant clock offset over the duration of the profiling window.

4.2.5 Bandwidth Calculation and Theoretical Limits

MESSAGE SIZE RECONSTRUCTION From the NCCL log metadata, the total message size is calculated as:

$$\text{MessageBytes} = \text{ElementCount} \times \text{sizeof(DataType)} \tag{4.9}$$

Data type sizes are defined by NCCL enumerations (e.g., `float16` = 2 bytes, `float32` = 4 bytes) as indicated in Table 4.3.

ALGORITHM BANDWIDTH The algorithm bandwidth represents the effective data rate from the application's perspective:

$$\text{BW}_{\text{algo}} = \frac{\text{MessageBytes}}{\text{Duration}_{\text{ns}}/10^9} \text{ [GB/s]} \tag{4.10}$$

BUS BANDWIDTH The bus bandwidth accounts for the actual data movement on the interconnect. For collective operations, the relationship between algorithm bandwidth and bus bandwidth depends on the operation type and algorithm used:

$$\text{BW}_{\text{bus}} = \text{BW}_{\text{algo}} \times \gamma \tag{4.11}$$

where γ is the bus factor that depends on the operation, algorithm, and communicator size N:

- **AllReduce (Ring):** $\gamma = \frac{2(N-1)}{N}$ (bidirectional ring traversal with reduction and broadcast phases)
- **AllReduce (Tree):** $\gamma = 2$ (data travels up and down the tree)
- **AllGather / ReduceScatter:** $\gamma = \frac{N-1}{N}$ (unidirectional ring)
- **Broadcast / SendRecv:** $\gamma = 1$ (direct transfer)

THEORETICAL BANDWIDTH CALCULATION The theoretical bandwidth represents the maximum achievable performance based on the hardware topology. The calculation process follows these steps:

1. **Topology Analysis:** From the NCCL topology logs, the analyzer extracts interconnect information including:
    - NVLink bandwidth between GPUs

- PCIe bandwidth to host
- System (SYS) bandwidth for indirect GPU connections
- Network (NET) bandwidth for inter-node communication

2. **Bottleneck Identification:** For each logical communicator, the analyzer determines the slowest link in the communication path:
   - For known communicators, where participating ranks are identified, all pairwise connections are analyzed and the minimum bandwidth is selected
   - For unknown communicators, a conservative estimate using the minimum of all available link types is used
   - The bottleneck is determined by:

$$\text{BW}_{\text{bottleneck}} = \min_{i,j \in \text{Comm}} \text{BW}_{\text{link}}(i, j) \tag{4.12}$$

   where the link bandwidth depends on whether GPUs are on the same node (NVLink/PCIe) or different nodes (Network)

3. **Theoretical Limits:** The theoretical bandwidth for algorithm and bus are calculated by applying collective-specific scaling factors to the bottleneck bandwidth:

$$\text{BW}_{\text{theo,algo}} = \text{BW}_{\text{bottleneck}} \times \alpha \tag{4.13}$$

$$\text{BW}_{\text{theo,bus}} = \text{BW}_{\text{bottleneck}} \times \beta \tag{4.14}$$

   where α and β are algorithm-specific factors:
   - **AllReduce (Ring):** $\alpha = \frac{N-1}{N}$, $\beta = \frac{2(N-1)}{N}$
   - **AllReduce (Tree):** $\alpha = 1$, $\beta = 2$
   - **AllGather / ReduceScatter:** $\alpha = \beta = \frac{N-1}{N}$
   - **Broadcast / Point-to-point:** $\alpha = \beta = 1$

Note that the theoretical bandwidth accounts for the fundamental limitation that in a ring algorithm with N participants, each GPU can only utilize $\frac{N-1}{N}$ of the link bandwidth due to the sequential nature of the algorithm.

EFFICIENCY METRICS The efficiency metrics quantify how close the observed performance is to the theoretical maximum:

$$\eta_{\text{algo}} = \frac{\text{BW}_{\text{algo}}}{\text{BW}_{\text{theo,algo}}} \times 100\% \tag{4.15}$$

$$\eta_{\text{bus}} = \frac{\text{BW}_{\text{bus}}}{\text{BW}_{\text{theo,bus}}} \times 100\% \tag{4.16}$$

These efficiency values indicate the degree to which the implementation achieves the hardware's theoretical limits.

4.2.6 Chrome Trace Export and Visualization

The analyzer outputs results in the Chrome Trace Event Format, enabling visualization in `chrome://tracing`. Each NCCL operation is represented as a duration event, with additional information:

Arguments
args
CUDA Kernel: ncclDevKernel_AllReduce_Sum_f16_RING_LL(ncclDevKernelArgsStorage<(unsigned long)4096>)
CUDA Grid: <<2, 1, 1>>
CUDA Block: <<544, 1, 1>>
Shared Mem (Static): 7104 bytes
Shared Mem (Dynamic): 82240 bytes
Registers Per Thread: 96
CUDA Stream ID: 31
CUDA Correlation ID: 511438
Local Memory: 280756224 bytes
Iteration: -1
Op Count (Hex): 0
Send Buffer: 0x1461be300000
Recv Buffer: 0x1461be300000
Root Rank: 0
Communicator Ptr: 0x558836a2b530
NCCL Stream Ptr: 0x55882ce051d0
Element Count: 2097152
Data Type: float16
Reduction Op: sum
Message Size (Bytes): 4194304
Algo BW (GB/s): 6.77055393774254
Bus BW (GB/s): 10.1558309066138
Bus Factor: 1.5
Theo
Algo BW (GB/s): 9.375
Bus BW (GB/s): 18.75
Bottleneck Link: Estimated (min: 12.5 GB/s)
NCCL Algorithm: ring
Efficiency (%): 72.2192420025871
Bus Efficiency: 54.1644315019403
Communicator Size: 4
Parallelism Type: Tensor Parallelism

Figure 4.2: Chrome Trace Event Information provided by the NCCL Trace Profiler

The visualization provides a hierarchical view of training performance, as shown in Figure 4.3:

- **Global Context (NVTX):** The top track (shared timeline) visualizes high-level training phases derived from NVTX markers, clearly delineating Iterations, Forward/Backward passes, and Microbatches.
- **Per-Rank Execution:** Each GPU rank is assigned a dedicated process group containing two synchronized threads:
  - *NCCL Operations:* The primary thread (top row per rank) displays the exact duration and sequence of collective kernels (e.g., `AllReduce`, `SendRecv`).
  - *Parallelism Classification:* A secondary thread (bottom row per rank) color-codes operations based on the detected parallelism strategy (e.g., DP, PP, TP), allowing for quick verification of communication patterns.
- **Performance Metadata:** Hovering over any operation reveals calculated metrics, including Algorithm/Bus Bandwidth, utilization efficiency relative to the theoretical hardware limit, and the identified bottleneck link (e.g., NVLink vs. PCIe).

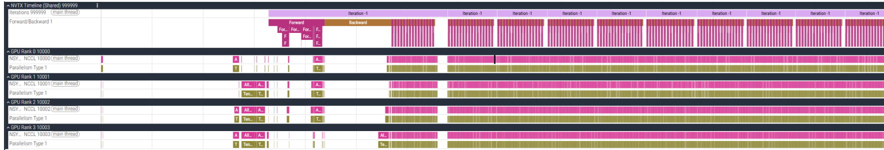


Figure 4.3: Chrome Trace Event Timeline produced by the NCCL Trace Profiler, showing the correlation between NVTX training steps (top), NCCL kernel execution (middle tracks), and parallelism classification (bottom tracks).

4.3 VALIDATION

The profiler was validated using a comprehensive validation suite that systematically tests all major components through unit and integration tests.

4.3.1 Alignment Algorithm Validation

The sequence alignment implementation is validated through five categories of tests that verify the scoring system produces correct correlations even under challenging conditions.

ASYMMETRIC PENALTY TEST This test validates that the harsh mismatch penalty prevents false positives by ensuring the algorithm prefers gaps over incorrect matches. A test sequence is constructed where one operation differs between the two sources:

- Nsys sequence: [`AllReduce`, `AllReduce`, `Broadcast`, `AllReduce`]
- NCCL sequence: [`AllReduce`, `AllReduce`, `Send`, `AllReduce`]

The third position contains `Broadcast` in Nsys but `Send` in NCCL logs, operations that are semantically different and should not be matched. With the configured scoring parameters (`MATCH_SCORE = +5`, `MISMATCH_PENALTY = -15`, `GAP_PENALTY = -5`), the algorithm must choose between:

1. Accepting the mismatch (score: -15)
2. Inserting two gaps (score: $2 \times -5 = -10$)

The test verifies that the algorithm selects option 2, resulting in zero mismatches in the final alignment. This demonstrates that the mismatch-to-gap penalty ratio of 7.5× successfully prevents false positives.

Result: The alignment produces 3 matches, 0 mismatches, and 2 gaps, confirming that the algorithm correctly avoids forcing incorrect correlations.

FALSE POSITIVE PREVENTION TEST This test validates that the alignment correctly identifies the matching sequence when presented with multiple similar but non-identical operation patterns. The scenario simulates a common challenge in production traces: NCCL logs may contain long sequences where similar patterns appear multiple times, and the algorithm must correlate Nsys operations to the correct matching segment rather than an incorrect similar segment.

A test case is constructed where Nsys contains a short sequence while NCCL logs contain three concatenated sequences with subtle variations:

- Nsys sequence: [`AllReduce`, `AllReduce`, `Broadcast`, `ReduceScatter`]
- NCCL sequence (three segments):
  - Segment 1: [`AllReduce`, `AllReduce`, `AllReduce`, `ReduceScatter`] (third op differs)
  - Segment 2: [`AllReduce`, `AllReduce`, `Broadcast`, `ReduceScatter`] (exact match)
  - Segment 3: [`AllReduce`, `Broadcast`, `AllReduce`, `ReduceScatter`] (order differs)

The alignment must identify that Nsys operations correspond to segment 2 of the NCCL logs, despite segments 1 and 3 being similar (75% overlap). This requires the scoring system to penalize mismatches heavily enough that the algorithm prefers the exact match over forcing alignment with the similar-but-different segments.

Result: The algorithm achieves $\geq 75\%$ of matches in the correct segment (segment 2), demonstrating that scoring parameters successfully distinguish between similar but non-identical sequences. This prevents false positives where operations would be incorrectly correlated to wrong segments that happen to share some common operation types.

CONSECUTIVE DUPLICATION TEST This test validates that the algorithm correctly handles scenarios where NCCL logs contain duplicate entries, each operation appearing twice consecutively, while Nsys traces contain only single instances.

A synthetic scenario is constructed to simulate this condition:

- Nsys sequence: [`AllReduce`, `Broadcast`, `AllGather`, `ReduceScatter`, `AllReduce`]
- NCCL sequence: [`AllReduce`, `AllReduce`, `Broadcast`, `Broadcast`, `AllGather`, `AllGather`, ...]

The NCCL sequence contains exactly twice the number of operations, with each operation type appearing in two consecutive entries. The test validates three critical properties:

Result: The alignment achieves 100% match rate (all 5 Nsys operations correctly matched), 0 mismatches, and a skip percentage of 50%, confirming that the scoring parameters allow the algorithm to gracefully handle consecutive duplication without degrading alignment quality. This demonstrates robustness to common data quality issues in production logging systems.

SEQUENCE BONUS TEST This test validates that consecutive matches of the same operation type receive appropriate bonuses, reinforcing correct alignments when operations occur in runs. A sequence containing operation runs is constructed:

$$[\texttt{AllReduce} \times 5, \texttt{Broadcast} \times 1, \texttt{ReduceScatter} \times 3]$$

Both Nsys and NCCL sequences are identical, and the algorithm should produce a perfect alignment. The sequence bonus (`SEQUENCE_BONUS = +1`) rewards consecutive matches, making it more favorable to maintain alignment through operation runs rather than introducing gaps.

Result: Perfect alignment is achieved, where all indices match: $i = j$ for all aligned pairs, confirming that the sequence bonus parameter correctly reinforces runs of identical operations.

ADAPTIVE GAP PENALTY TEST This test validates that consecutive gaps incur progressively higher penalties, discouraging scattered alignments in favor of contiguous matching. A scenario is created where gaps are necessary but their placement matters:

- Nsys: [`AllReduce` × 5]
- NCCL: [`AllReduce`, `Broadcast`, `Send`, `Recv`, `AllReduce` × 3]

The three mismatched operations (`Broadcast`, `Send`, `Recv`) must be handled as gaps. With adaptive gap penalties (`GAP_RUN_MULTIPLIER = 1.3`), consecutive gaps become increasingly expensive: first gap costs -5, second gap costs -9.88, third gap costs -12.844. The algorithm should prefer scattering these gaps them throughout the alignment rather than grouping them together.

Result: Gap analysis shows a maximum consecutive gap run of ≤ 2, with gaps distributed across multiple short runs rather than one long run. This confirms that the adaptive penalty successfully discourages long gap sequences while still allowing necessary corrections.

4.3.2 Integration Testing: Comparative Scenarios

The validation suite includes end-to-end integration tests that compare the optimized Needleman-Wunsch alignment against a baseline sliding window heuristic across four scenarios. These tests use synthetic

multi-rank workloads with known ground truth, enabling precise measurement of alignment accuracy.

Each scenario simulates 4 ranks with 200 operations each, systematically varying the presence of dropped operations to test robustness:

1. **Easy 1:1**: No dropped operations, perfect match
2. **Nsys Drops 20%**: Simulates incomplete Nsys traces
3. **NCCL Drops 20%**: Simulates incomplete NCCL logs
4. **Both Drop 20%**: Worst-case scenario with incomplete data from both sources

For each scenario, the validation suite measures precision, recall, and F1 score by comparing aligned pairs against ground truth. Given a set of aligned pairs produced by the algorithm and a set of ground-truth aligned pairs, let:

- TP (true positives) be the number of correctly aligned pairs,
- FP (false positives) be the number of incorrectly aligned pairs produced by the algorithm,
- FN (false negatives) be the number of ground-truth pairs that the algorithm failed to align.

Precision, recall, and F1 score are then defined as:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{4.17}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{4.18}$$

$$F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4.19}$$

These metrics quantify the accuracy of the alignment algorithm in terms of correctness (precision), completeness (recall), and their harmonic mean (F1 score).

Table 4.5 shows that the optimized Needleman-Wunsch alignment significantly outperforms the baseline sliding window heuristic in challenging scenarios where NCCL logs contain missing operations. In the balanced case where both data sources are complete, the sliding window achieves perfect 100% F1 score while the optimized approach achieves 98.8%, a marginal 1.2% degradation. Similarly, when only Nsys drops 20% of operations, both methods perform comparably (91.6% vs 91.2%).

Table 4.5: Comparative Validation Results: F1 Scores Across Scenarios

Scenario	Sliding Window	Optimized NW	Improvement
Easy 1:1	1.000	0.988	−1.2%
Nsys Drops 20%	0.916	0.912	−0.4%
NCCL Drops 20%	0.276	0.868	+214.8%
Both Drop 20%	0.256	0.805	+214.0%
Average	**0.612**	**0.893**	**+46.0%**

However, the advantage of the optimized approach becomes dramatic when NCCL logs are incomplete. When logs drop 20% of operations, the sliding window's F1 score collapses to 27.6% while the optimized approach maintains 86.8%, a 214.8% improvement. This stark difference arises because the sliding window method matches the first available log entry within its search window; when log entries are missing, it systematically matches to incorrect subsequent operations. The Needleman-Wunsch algorithm, by contrast, uses dynamic programming to find the globally optimal alignment, allowing it to correctly skip over gaps in the log sequence.

The worst-case scenario, where both sources drop 20% of operations, further demonstrates this robustness: the optimized approach achieves 80.5% F1 while the baseline falls to 25.6%, a 214.0% improvement.

The average F1 score across all scenarios is 0.893 for the optimized approach versus 0.612 for sliding window, representing a 46.0% improvement in overall alignment accuracy. This validates that while the sliding window heuristic performs well on complete data, the dynamic programming approach with carefully tuned scoring parameters is essential for handling real-world data quality issues.

Pass Criteria: The integration tests pass if the optimized alignment achieves $\geq$ 85% average F1 score and outperforms the baseline on scenarios with incomplete data. Current results exceed this threshold with an average F1 of 89.3%.

4.3.3 Component-Level Tests

Beyond alignment validation, the suite tests individual profiler components to ensure correct operation of supporting infrastructure.

TOPOLOGY PARSING The topology parser is validated by processing synthetic NCCL debug logs containing the `=== System : maxBw X.X totalBw Y.Y ===` header and nested connectivity information. The test verifies:

- Extraction of bus IDs and bidirectional mapping to local ranks

- Parsing of link types (PCI, NVLink, NVSwitch, Network) and associated bandwidths
- Correct handling of multi-socket NUMA configurations
- Selection of the first topology block when multiple blocks are present in logs

Result: Bus ID extraction achieves 100% accuracy on test logs, with correct identification of GPU-to-GPU NVLink connections (80.0 GB/s), PCIe links to CPU (24.0 GB/s), and network interfaces (12.5 GB/s).

COMMUNICATOR INFERENCE The communicator inference module is tested with synthetic workloads where ranks are organized into known logical groups (e.g., 2 TP groups of 2 ranks each). Each group performs an identical sequence of operations with distinct communicator pointers. The test validates that ranks with matching operation sequences are correctly grouped and assigned consistent communicator names (e.g., `TP_Group_0`, `TP_Group_1`).

Result: Communicator inference achieves 100% accuracy on test workloads with 2-4 logical groups, correctly identifying rank membership based on operation sequence similarity.

CLOCK SYNCHRONIZATION Clock synchronization is validated using synthetic traces with known artificial clock offsets between ranks. The validation creates multi-rank workloads where each rank's timestamps are shifted by a known offset (e.g., rank 0: 0 ms, rank 1: +50 ms, rank 2: -30 ms). The median-based offset calculation is expected to recover these offsets within 200 ns tolerance.

Result: Clock offset calculation achieves < 200 ns accuracy on synthetic workloads, validating the median-based approach for temporal alignment across nodes.

5 LLM COMMUNICATION ANALYSIS

5.1 EXPERIMENTAL INFRASTRUCTURE: NODE ARCHITECTURES

To evaluate the communication patterns of Megatron-LM across diverse hardware configurations, experiments are conducted on two distinct node architectures spanning two generations of NVIDIA datacenter GPUs. These architectures represent a progression in interconnect technology, from PCIe-only designs to advanced NVSwitch fabrics, each with different communication characteristics and performance implications for distributed training. This section provides detailed topology descriptions for each system as extracted from NCCL debug logs and using the `nvidia-smi topo -m` command.

5.1.1 PCIe-Only Architecture: NVIDIA A40

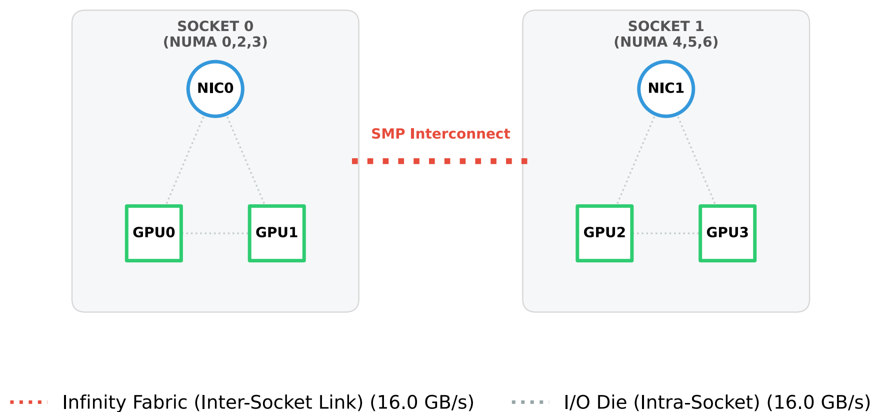


Figure 5.1: Topology map of the NVIDIA A40 PCIe-only architecture

The NVIDIA A40 node, illustrated in Figure 5.1, represents the baseline architecture: four NVIDIA A40 GPUs connected exclusively via PCIe, with no NVLink connectivity. NCCL reports a maximum bandwidth of approximately 24 GB/s across all four GPUs, but this is reduced to 16 GB/s due to the NUMA configuration of this node, as each GPU is in their own NUMA subdomain, meaning all GPU communication has to cross the I/O die of the CPUs. This is confirmed

by the topology matrix, which shows all inter-GPU paths classified as SYS, indicating that communication must traverse the host CPU's interconnect. The node is equipped with 2× HDR100 InfiniBand NICs for multi-node communication, though the same NUMA partitioning limits effective bandwidth to 16 GB/s per GPU for network traffic. This architecture is representative of systems designed for inference workloads or independent parallel tasks, where inter-GPU communication is minimal. For distributed training, this limited bandwidth creates significant bottlenecks.

5.1.2 Full NVLink Mesh: NVIDIA A100-SXM4 (40GB)

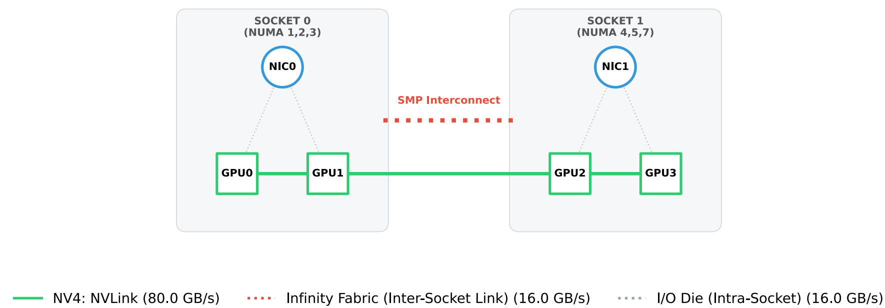


Figure 5.2: Topology map of the NVIDIA A100-SXM4 (40GB) full NVLink mesh architecture

The NVIDIA A100-SXM4 (40GB) node in Figure 5.2 transitions to a full NVLink mesh topology. All four GPUs have direct NV4 connections to every other GPU, eliminating PCIe and host CPU traversal for inter-GPU communication. Each NVLink connection provides 80 GB/s of bidirectional bandwidth, and NCCL measures approximately 80 GB/s effective bandwidth for collective operations, a 5× improvement over the A40 baseline. For multi-node communication, the node is equipped with 2× HDR200 InfiniBand NICs, providing substantially higher network bandwidth compared to the A40 configuration.

5.1.3 Architectural Implications

These two architectures span a performance range from 16 GB/s (A40) to 80 GB/s (A100), a 5× difference in communication bandwidth. This diversity enables communication pattern observation under varying communication-to-computation ratios and bottleneck characteristics. For instance, the A40's PCIe-limited bandwidth makes communication the dominant bottleneck, while the A100's full NVLink mesh shifts bottlenecks toward computation or cross-node networking. Understanding these architectural differences is essential for interpreting the profiling traces analyzed in subsequent sections.

5.1.4 Hardware Characterization and Baselines

To establish performance baselines and validate the hardware topology descriptions provided in Section 5.1, NCCL bandwidth benchmarks were conducted across all experimental platforms. These measurements characterize the achievable bandwidth for each collective operation type as a function of message size, revealing the fundamental communication limits imposed by each architecture's interconnect topology.

The benchmarks employ the official NCCL tests suite [17], which measures collective operation throughput by averaging performance over multiple iterations at each message size. Each test reports bus bandwidth, the effective bidirectional data movement rate that accounts for the algorithmic complexity of the collective operation as explained in section 4.2.5.

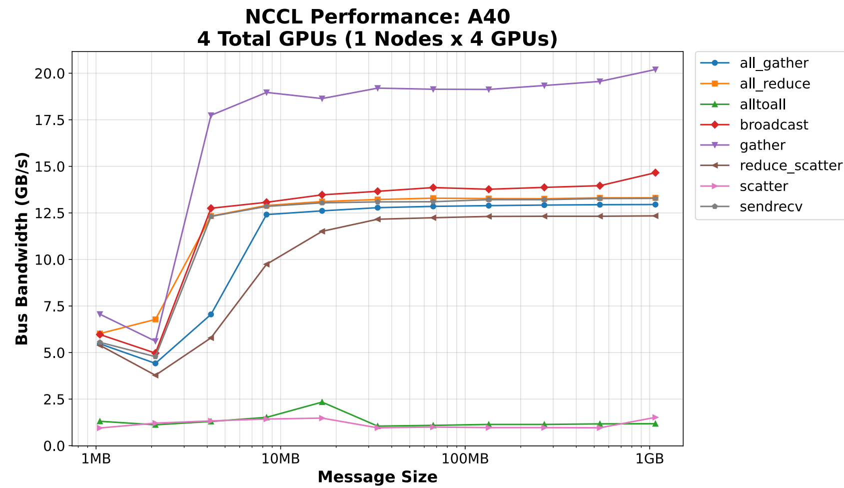
Figures 5.3a and 5.3b present bandwidth measurements for message sizes ranging from 1 MB to 1 GB across two representative configurations: single-node A40 (4 GPUs) and single-node A100 (4 GPUs). The results quantify the architectural differences discussed in Section 5.1 and establish the expected communication performance for each system.

KEY OBSERVATIONS FROM BANDWIDTH MEASUREMENTS

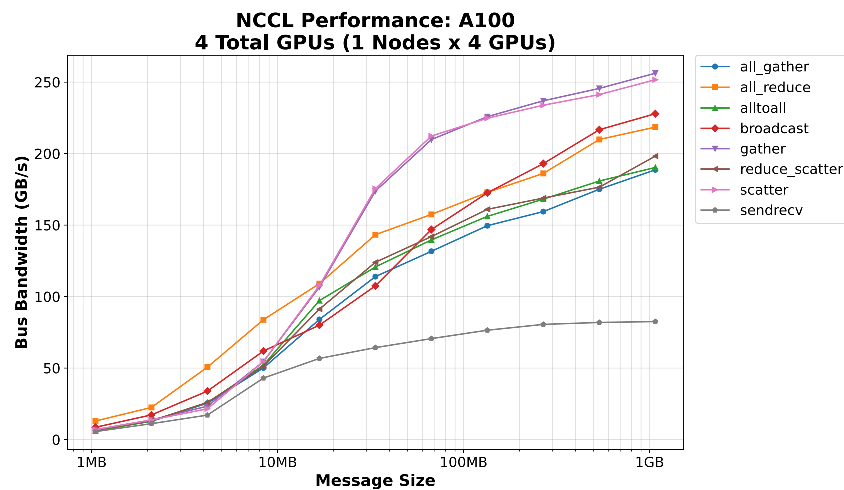
MESSAGE SIZE DEPENDENCY All platforms exhibit strong message size dependency, with bandwidth increasing logarithmically until reaching a saturation point determined by the underlying interconnect. For PCIe-based systems (A40), saturation occurs around 10 MB, while the NVLink system (A100) continues scaling until 100-500 MB. This behavior reflects the latency-bandwidth tradeoff: small messages are dominated by protocol overhead and launch latency, while large messages amortize these costs and approach hardware limits.

COLLECTIVE OPERATION HIERARCHY The bandwidth hierarchy across collective types remains consistent across architectures, though absolute values scale with interconnect capability. `AllReduce`, `AllGather`, and `ReduceScatter` operations achieve comparable throughput, typically 10-12 GB/s on A40 and 190-220 GB/s on A100. In contrast, `alltoall` exhibits significantly lower bandwidth: 1-2 GB/s on A40 and approximately 190 GB/s on A100. This reflects `AlltoAll`'s all-to-all communication pattern, which creates more contention on the interconnect fabric than reduce-based collectives.

ASYMMETRIC OPERATIONS `Gather` and `Scatter` operations, which concentrate data movement to/from a single root rank, achieve the highest bandwidth on all platforms. On A40, `Gather` peaks at 22 GB/s (nearly 2$\times$ higher than `AllReduce`), exploiting the star topology where



(a) NVIDIA A40 single-node configuration (4 GPUs). The PCIe-only topology limits most collective operations to approximately 12 GB/s, with the exception of `Gather`, which peaks near 22 GB/s due to its asymmetric communication pattern that concentrates bandwidth on a single root GPU.



(b) NVIDIA A100-SXM4 (40GB) single-node configuration (4 GPUs). The full NVLink mesh topology enables dramatically higher bandwidth: `Gather` and `Scatter` operations exceed 250 GB/s at large message sizes, while `AllReduce` and `AllGather` achieve approximately 190-220 GB/s, representing a 15-20× improvement over the A40 baseline.

Figure 5.3: NCCL bandwidth characterization comparing single-node 4-GPU configurations. (a) A40 with PCIe-only topology achieves ∼12 GB/s for most operations, while (b) A100-SXM4 with full NVLink mesh topology delivers 15-20× higher bandwidth, demonstrating the dramatic impact of interconnect architecture on collective operation performance.

all GPUs concurrently send to the root without inter-sender contention. On A100, `Gather/Scatter` exceed 250 GB/s, demonstrating that the NVLink mesh provides sufficient bisection bandwidth to support concurrent all-to-one data flow without significant congestion.

5.2 EXPERIMENTAL METHODOLOGY

Having established the hardware infrastructure and baseline performance characteristics, this section describes the experimental design used to systematically characterize collective communication patterns across diverse parallelism strategies.

5.2.1 Model Architectures

Experiments utilize two model configurations specifically designed for communication pattern analysis rather than large-scale training. This constraint does not limit the generalizability of results, as communication patterns are determined by parallelism strategy and not by absolute model size. These compact architectures enable complete profiling of entire training runs within the memory and time constraints imposed by Nsys, while still exhibiting the collective communication patterns characteristic of production-scale LLMs, only the frequency of communications and message sizes differ.

DENSE TRANSFORMER MODEL (`dense`) The baseline dense model, designated `dense`, implements a GPT-style decoder-only transformer with the following architecture:

- **Layers:** 16 transformer blocks
- **Hidden Dimension:** 512
- **Attention Heads:** 8
- **Sequence Length:** 1024 tokens
- **Vocabulary Size:** 50,257 (GPT-2 tokenizer)
- **Total Parameters:** Approximately 50.3 million
- **Precision:** FP16 mixed precision (FP32 master weights)

MIXTURE-OF-EXPERTS MODEL (`dense_moe`) To characterize MoE communication patterns, particularly the `AlltoAll` exchanges central to expert parallelism, a sparse model variant was developed:

- **Base Architecture:** Identical to `dense` (16 layers, 512 hidden dim, 8 heads)

- **Expert Configuration:** 4 experts per MoE layer
- **MoE Layer Frequency:** Every 2nd transformer layer (layers 1, 3, 5, 7)
- **Expert Capacity Factor:** 1.0 (no token dropping)
- **Router:** Top-1 gating with softmax routing
- **Precision:** BF16 (required for Megatron's grouped GEMM kernels)

The Megatron-LM implementation imposes specific constraints on MoE models that directly impact model configuration. First, `-bf16` precision is mandatory due to the grouped GEMM operations used for expert computation; FP16 is not supported. Second, bias terms must be disabled via `-disable-bias-linear` as bias is incompatible with the expert implementation. Third, EP degree must evenly divide the number of experts (valid configurations for 4 experts: EP=1, 2, 4). Finally, PP>1 frequently triggers NCCL timeout errors in MoE models, possibly stemming from interaction between the PP and MoE communication patterns in the current setup, necessitating PP=1 for all MoE experiments.

5.2.2 Training Configuration and Hyperparameters

All experiments employ consistent training hyperparameters to isolate the impact of parallelism strategies on communication patterns:

- **Global Batch Size:** 64 samples
- **Training Iterations:** 10 complete iterations
- **Learning Rate:** 6.0×10^{-5} with cosine decay
- **Warmup Fraction:** 0.001
- **Optimizer:** AdamW ($\beta_1 = 0.9$, $\beta_2 = 0.95$, weight decay=0.1)
- **Gradient Clipping:** 1.0
- **Random Seed:** 42 (fixed for reproducibility)

The micro-batch size is dynamically calculated as:

$$\text{MBS} = \frac{\text{Global Batch Size}}{\text{DP} \times \text{Gradient Accumulation Steps}} \tag{5.1}$$

where the number of gradient accumulation steps is chosen to fit within GPU memory constraints while maintaining consistent global batch size across configurations. Sequence Parallelism (SP) is enabled automatically when $\text{TP} > 1$ via the `-sequence-parallel` flag, as per

Megatron-LM best practices. For configurations with TP=1, SP is disabled as it provides no benefit without tensor partitioning.

All experiments use the FineWeb dataset [60], a large-scale, high-quality web corpus designed for language model pretraining, split into training (94.9%), validation (5.0%), and test (0.1%) partitions.

5.2.3 Distributed Training Framework

All experiments use NVIDIA Megatron-LM (commit `73a28a1`, release `core_r0.13.0`). The training stack consists of:

- PyTorch 2.7.1 with CUDA 12.8
- NCCL 2.28.9.1
- cuDNN 9.10
- NVIDIA Apex (commit `bfb500c`)
- Transformer Engine 2.5.0
- Flash Attention v2.7.4

All components are compiled with CUDA extensions enabled for optimized kernel execution. To support heterogeneous GPU platforms, separate conda environments are maintained for A100 and A40 systems with architecture-specific CUDA compilation flags (`TORCH_CUDA_ARCH_LIST="8.0"` for A100/A40). For the full environment setup script see Appendix a.2.

5.2.4 Execution

Training is launched using PyTorch's `torchrun` utility for robust multi-rank coordination. The experiments are run using `torchrun` directly on the allocated node. The launcher script, see Appendix a.3, handles parameter validation, micro-batch size calculation, and dynamic SP configuration based on the selected TP degree. Parallelism configurations are selected to isolate dominant communication patterns. On the 4-GPU A40 node and the 4-GPU A100 node pure parallelism strategies are tested, i.e. DP=4, PP=4, TP=4, and EP=4.

5.3 COMMUNICATION PATTERN ANALYSIS

This section analyzes the NCCL communication patterns observed across the aforementioned parallelism configurations. The analysis focuses on characterizing the Communication traffic signatures: operation types, frequencies, message sizes, and temporal distributions, to validate theoretical predictions and identify LLM communication characteristics.

BANDWIDTH VALIDATION For matched operations, the profiler's per-operation bandwidth calculations align with the hardware baselines established in Section 5.2: individual collectives achieve bandwidth consistent with the NCCL benchmark results shown in Figures 5.3a and 5.3b, confirming that the timing correlation produces accurate performance measurements. The following analysis focuses on communication structure and volume characteristics rather than systematic bandwidth comparison, the aggregate volume measurements are derived directly from NCCL debug logs to ensure completeness.

INITIALIZATION AND SHUTDOWN COMMUNICATION All presented measurements include the complete training lifecycle: initialization communication before the first iteration, steady-state training communication during iterations, and shutdown communication after the final iteration. The initialization phase typically includes framework setup, buffer allocation, communicator establishment, and initial parameter synchronization, while shutdown involves final checkpointing, metric aggregation, and cleanup operations. The specific initialization and shutdown patterns are parallelism-strategy-dependent, and these auxiliary communication phases are visible in the operation counts and NCCL Trace Profiler timelines, contributing to the characteristic signatures of each parallelism strategy.

5.3.1 Data Parallelism (DP=4)

Figure 5.4 presents the observed NCCL operation distribution over 10 training iterations. Both A40 and A100 nodes exhibit identical counts: 220 `AllReduce` (71.4%), 84 `AllGather` (27.3%), and 4 `Broadcast` (1.3%).

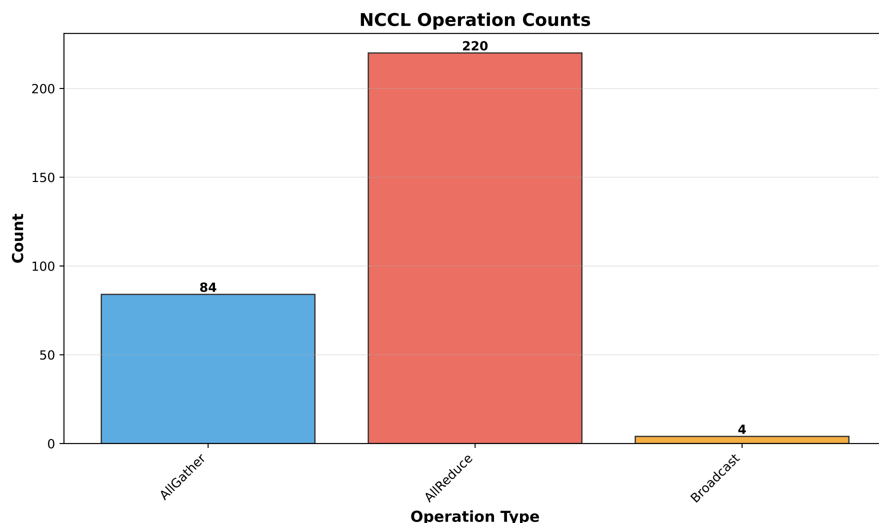


Figure 5.4: NCCL operation counts for DP=4 configuration showing `AllReduce` dominance (71.4%). The A100 exhibits identical counts, confirming architecture-invariant patterns.

The AllReduce-dominated pattern aligns with theoretical expectations for gradient synchronization presented in Section 2.4.1.

Figure 5.5 presents the volume breakdown by collective type, confirming the dominance of gradient synchronization in standard DP. AllReduce operations account for 5.72 GB (89.8% of total volume), handling gradient synchronization across all model parameters. Broadcast operations contribute 0.64 GB (10.0%), likely for metadata or learning-rate schedule distribution. AllGather contributes essentially nothing at below 10 MB. The overwhelming AllReduce dominance quantifies the core DP communication pattern: aggregating gradients across replicas is the primary network operation, with minimal auxiliary communication.

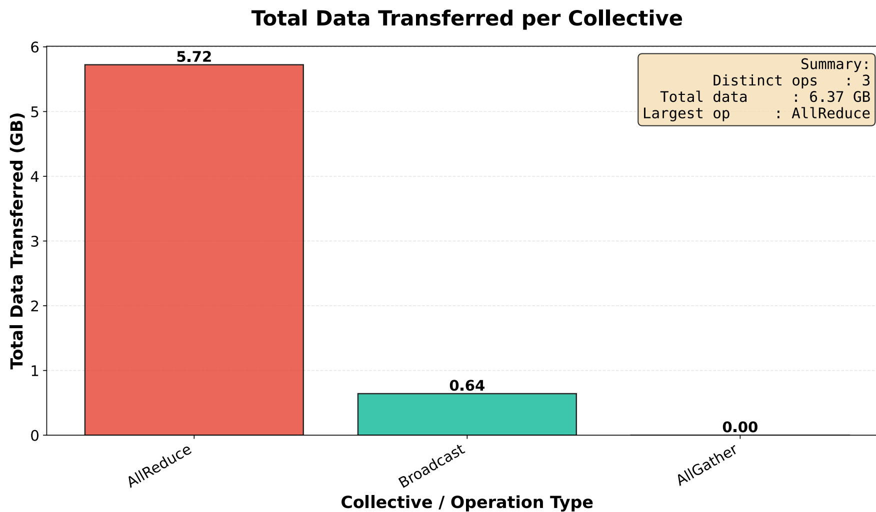


Figure 5.5: Data volume per collective type for DP=4 showing AllReduce dominance (5.72 GB, 89.8%) for gradient synchronization, with minor Broadcast contribution (0.64 GB, 10.0%). Total: 6.37 GB.

TEMPORAL COMMUNICATION PATTERN The NCCL Trace Profiler timeline in Figure 5.6 confirms that gradient synchronization occurs exclusively at iteration boundaries rather than progressively during the backward pass. All operations are concentrated at the end of each iteration, after all layers have completed their gradient computation, appearing as synchronised bursts across all GPU ranks. This bursty structure, brief communication windows separated by long computation-only intervals, is the characteristic temporal signature of DP training.

IMPACT OF DISTRIBUTED OPTIMISER (ZERO-1) To assess the communication impact of optimiser-state sharding, the DP=4 configuration is also evaluated with Megatron-LM's distributed optimiser enabled (`--use-distributed-optimizer`), which implements ZeRO-1 [65] by partitioning optimiser states across data-parallel ranks.

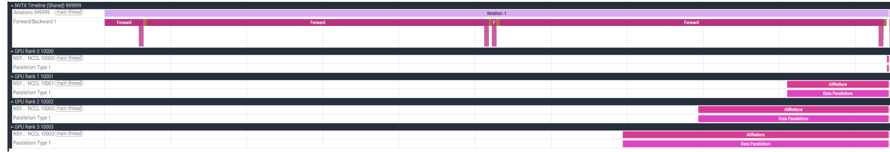


Figure 5.6: NCCL Trace Profiler timeline for DP=4 showing operations concentrated at iteration boundaries across all GPU ranks.

Figure 5.7 reveals a fundamental shift in operation composition: 220 `AllReduces` (63.2%), 84 `AllGathers` (24.1%), 40 `ReduceScatters` (11.5%, new), and 4 `Broadcasts` (1.1%).

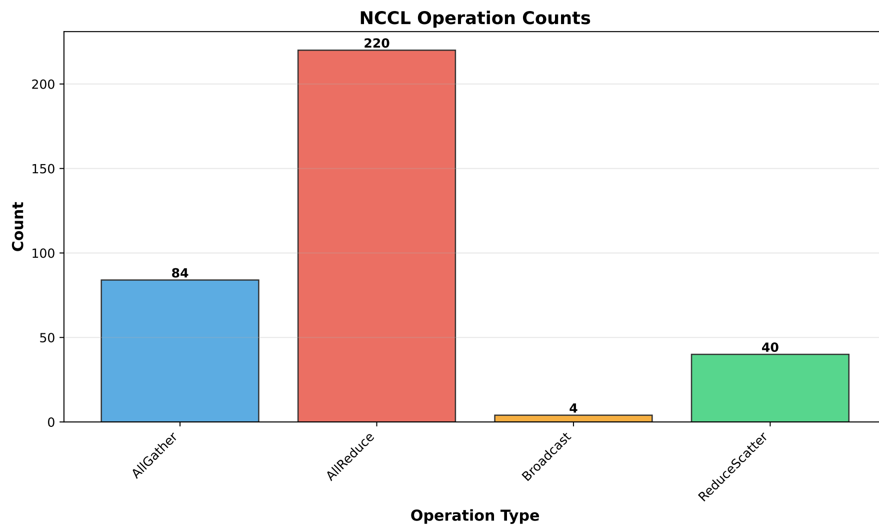


Figure 5.7: NCCL operation counts for DP=4 with distributed optimiser showing introduction of `ReduceScatter` operations (11.5%) alongside the traditional `AllReduce` pattern. Total operations increase from 308 to 348.

The appearance of 40 `ReduceScatter` operations (1 per iteration, per rank) is the key signature of ZeRO-1 optimization. As illustrated in Figure 5.8, the distributed-optimiser workflow replaces the standard gradient `AllReduce` pattern with a more complex four-step process: first each GPU computes gradients locally; then `ReduceScatter` reduces and partitions gradients across ranks, with each rank receiving only its assigned shard; after that each rank performs the optimiser update on its local parameter shard; and finally `AllGather` reconstructs the full parameter tensors by gathering shards from all ranks before the next forward pass.

The 40 `ReduceScatter` operations observed correspond to step 2 in the workflow, while the 84 `AllGather` operations correspond to step 4.

Analysing volume by collective type (Figure 5.9) reveals a shift in distribution compared to baseline DP, reflecting the efficiency of parameter sharding. `ReduceScatter` operations now dominate with 1.43 GB (68.8% of total volume), handling the sharded gradient re-

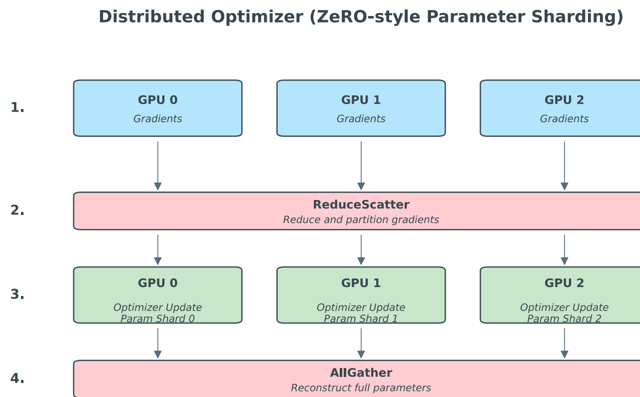


Figure 5.8: Distributed optimiser workflow showing ZeRO-1 parameter sharding. After local gradient computation, `ReduceScatter` partitions reduced gradients across ranks. Each GPU updates only its parameter shard, then `AllGather` reconstructs complete parameters for the next iteration.

duction that replaces most of baseline's `AllReduce` traffic. `Broadcast` operations remain unchanged at 0.64 GB (30.8%). `AllReduce` volume drops to essentially zero (0.01 GB, 0.5%), and `AllGather` contributes negligibly. The total volume reduction from 6.37 GB to 2.08 GB (67% decrease) quantifies the efficiency gain from ZeRO-1 sharding: by partitioning optimizer states and only reducing gradients to their assigned shards, the distributed optimizer eliminates the redundant full-gradient replication inherent in standard DP's `AllReduce` pattern.

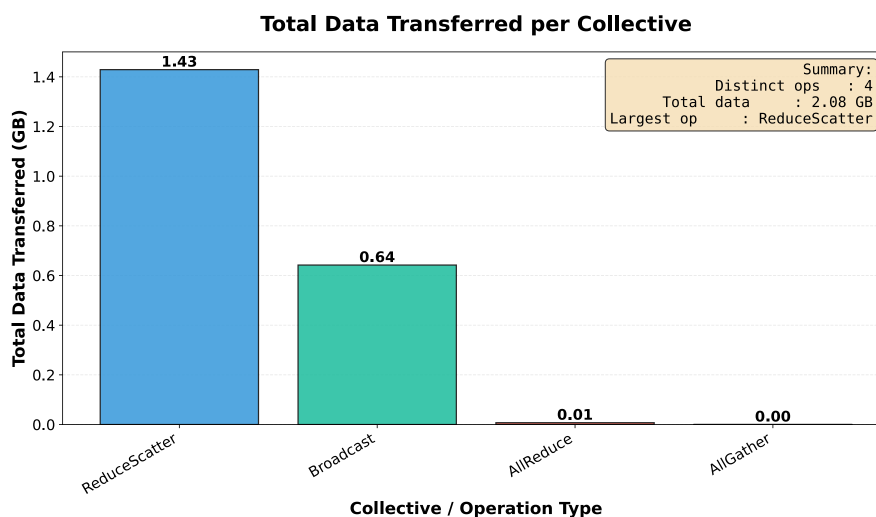


Figure 5.9: Data volume per collective type for DP=4 with distributed optimiser showing `ReduceScatter` dominance (1.43 GB, 68.8%) and dramatic total volume reduction (2.08 GB vs. 6.37 GB baseline, 67% decrease). The shift from `AllReduce` to `ReduceScatter` quantifies ZeRO-1's efficiency through sharded gradient reduction.

The NCCL Trace Profiler timeline in Figure 5.10 shows the transformed communication pattern with distributed optimiser enabled. Like standard DP, communication is concentrated at iteration boundaries, but now involves a mix of `AllReduce`, `AllGather`, and `ReduceScatter` operations.

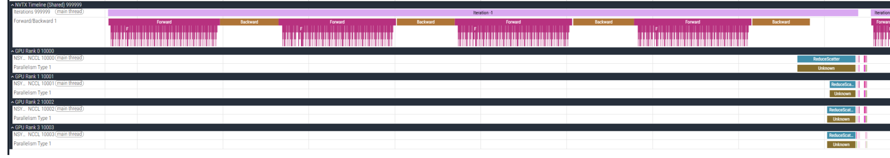


Figure 5.10: NCCL Trace Profiler timeline for DP=4 with distributed optimiser showing an operation mix with `AllReduce`, `AllGather`, and `ReduceScatter` at iteration boundaries.

COMMUNICATION SIGNATURE SUMMARY DP exhibits a distinct Communication signature: collective-dominated traffic with 71% `AllReduce` operations, an extremely bursty temporal pattern with communication concentrated in brief periodic windows separated by long computation-only gaps, and architecture-invariant operation patterns with identical counts across different interconnect technologies.

With the distributed optimiser, DP exhibits modified characteristics: introduction of `ReduceScatter` operations (11.5%), an operation-count increase (+13%, from 308 to 348), and retention of the bursty temporal pattern with communication still concentrated at iteration boundaries.

5.3.2 Pipeline Parallelism (PP=4)

Figure 5.11 shows a fundamentally different operation mix from DP: 1 080 `SendRecv` (72.4%), 322 `AllReduce` (21.6%), 84 `AllGather` (5.6%), and 4 `Broadcast` (0.3%). The `SendRecv` dominance reflects PP's point-to-point communication model.

With this 4-stage pipeline each GPU hosts one stage, creating 3 communication boundaries between consecutive stages. The 1 080 `SendRecv` operations per 10 iterations represent 108 operations per iteration, which translates to 36 transfers per boundary ($108 \div 3$), accounting for the bidirectional exchange of forward-pass activations and backward-pass gradients across each boundary as microbatches progress through the 1F1B schedule.

Breaking down communication volume by operation type for PP=4 (Figure 5.12) reveals a more balanced distribution compared to DP, reflecting the heterogeneous nature of pipeline communication. `Send` operations (which include paired `SendRecv` transfers) account for 3.75 GB (46.9% of total volume), representing activation forwarding and gradient backpropagation between pipeline stages. `AllReduce` operations contribute 2.97 GB (37.2%), presumably handling periodic gradient

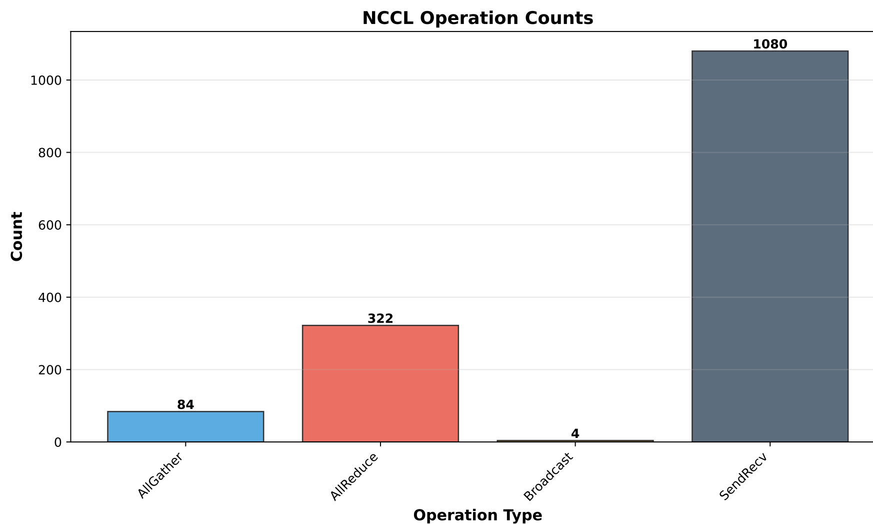


Figure 5.11: NCCL operation counts for PP=4 showing SendRecv dominance (72.4%). Both A40 and A100 exhibit identical counts.

synchronization across the pipeline for shared parameters. `Broadcast` operations transfer 1.27 GB (15.9%), likely for learning-rate schedules, loss values, or coordination signals. `AllGather` contributes negligibly at below 10 MB.

TEMPORAL COMMUNICATION PATTERN The NCCL Trace Profiler timeline in Figure 5.13 reveals asymmetric communication distribution across stages: middle stages Ranks 1 and 2 exhibit the densest `SendRecv` activity, while first/last stages communicate primarily in one direction. Unlike DP's communication concentrated at iteration boundaries, PP exhibits continuous `SendRecv` activity throughout training as microbatches flow steadily through the pipeline, with no large idle gaps or synchronization barriers during steady-state execution. Periodic `AllReduce` operations appear at regular intervals, possibly for global metadata synchronization, loss aggregation, or pipeline coordination between flush cycles.

IMPACT OF VIRTUAL PIPELINE PARALLELISM To assess the impact of virtual-pipeline optimization on communication patterns, the PP=4 configuration is evaluated with virtual-pipeline parallelism enabled (`--num-layers-per-virtual-pipeline-stage 2`), which interleaves 2 virtual stages per physical stage to reduce pipeline bubbles, as depicted in Figure 5.14.

Figure 5.15 shows a dramatic increase in communication frequency: 4 480 `SendRecvs` (91.5%), representing 448 per iteration, a 4.1× increase over baseline, along with 334 `AllReduces` (6.8%), 84 `AllGathers` (1.7%), and 8 `Broadcasts` (0.2%). Total operations increase from 1 490 to 4 906.

The 4.1× increase in `SendRecv` operations, from 108 to 448 per iteration, occurs because each physical stage now processes 2 vir-

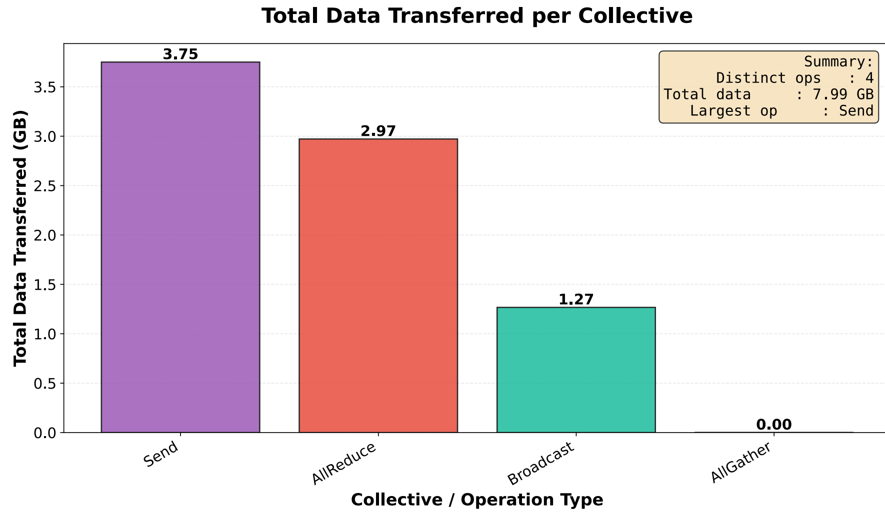


Figure 5.12: Data volume per collective type for PP=4 showing balanced distribution: `Send` (3.75 GB, 46.9%), `AllReduce` (2.97 GB, 37.2%), and `Broadcast` (1.27 GB, 15.9%). The balance reflects PP's dual communication pattern: continuous stage-to-stage transfers plus periodic global synchronization. Total: 7.99 GB.

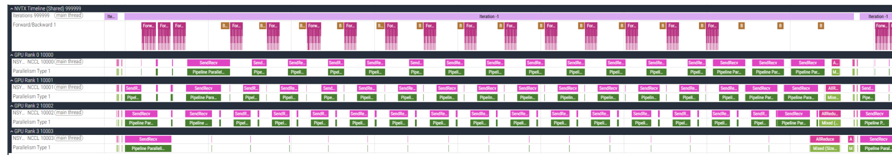


Figure 5.13: NCCL Trace Profiler timeline showing asymmetric communication across pipeline stages, with middle stages exhibiting the densest activity.

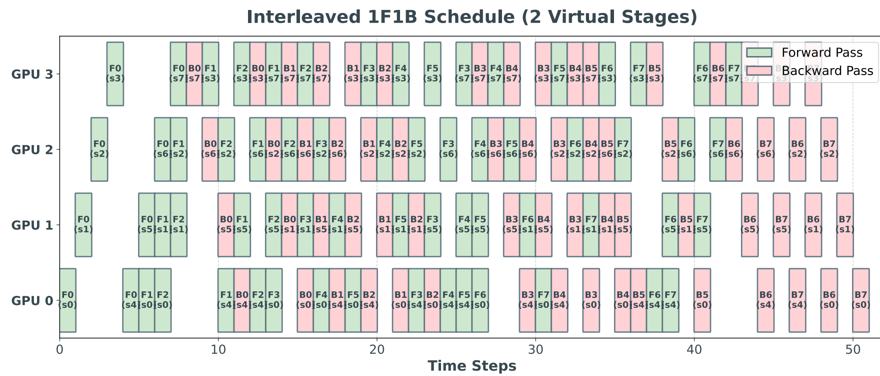


Figure 5.14: Interleaved 1F1B pipeline schedule with 2 virtual stages across 4 GPUs. The diagram illustrates the execution timeline of forward passes (F0-F7) and backward passes (B0-B7) for 8 microbatches (s0-s7). Each GPU processes multiple virtual pipeline stages in an interleaved pattern, enabling better GPU utilization by overlapping computation and reducing pipeline bubbles compared to traditional 1F1B scheduling.

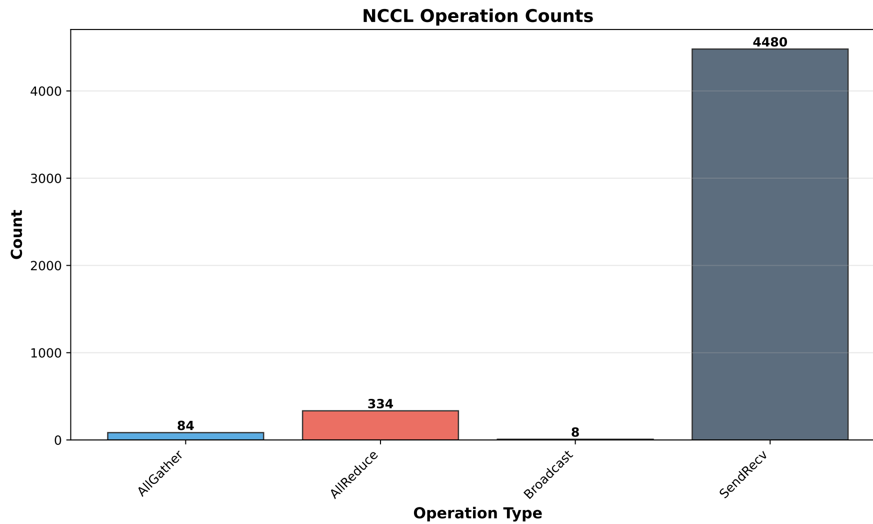


Figure 5.15: NCCL operation counts for PP=4 with virtual pipeline (2 stages per physical stage) showing a 4.1× increase in `SendRecv` operations (91.5%) compared to baseline. Total operations increase from 1 490 to 4 906, with `SendRecv` dominance rising from 72.4% to 91.5%.

tual stages, with each virtual stage containing 2 layers (16 layers ÷ 4 physical stages ÷ 2 virtual stages = 2 layers per virtual stage), as illustrated in Figure 5.14. The interleaved schedule requires more frequent activation- and gradient-exchanges between virtual stages, as microbatches alternate between the two virtual stages within each physical rank to maintain pipeline flow and minimize bubbles.

The volume implications of the operation-count explosion become apparent when examining collective-type contributions with virtual pipelining (Figure 5.16). `Send` operations surge to 8.75 GB (61.4% of total volume), representing a 133% increase from baseline's 3.75 GB due to the 4.1× increase in stage-to-stage transfers. `AllReduce` volume remains unchanged at 2.97 GB (20.8%), confirming that global synchronization requirements are independent of virtual-stage configuration. `Broadcast` operations double to 2.53 GB (17.7%) from baseline's 1.27 GB, likely reflecting increased coordination overhead for the more complex interleaved schedule. `AllGather` remains negligible. The total volume increase from 7.99 GB to 14.26 GB (78% increase) primarily stems from the `Send` explosion, quantifying the communication cost of virtual pipelining: more frequent activation/gradient exchanges between interleaved virtual stages drive the volume increase, while the unchanged `AllReduce` volume confirms that virtual staging affects only stage-to-stage communication, not global synchronization.

Despite this substantial communication overhead with operations increasing 3.3×, training time remains essentially unchanged (A40: 21 s baseline vs. 20 s VPP; A100: 18 s baseline vs. 18 s VPP). This demonstrates that virtual pipelining successfully trades communica-

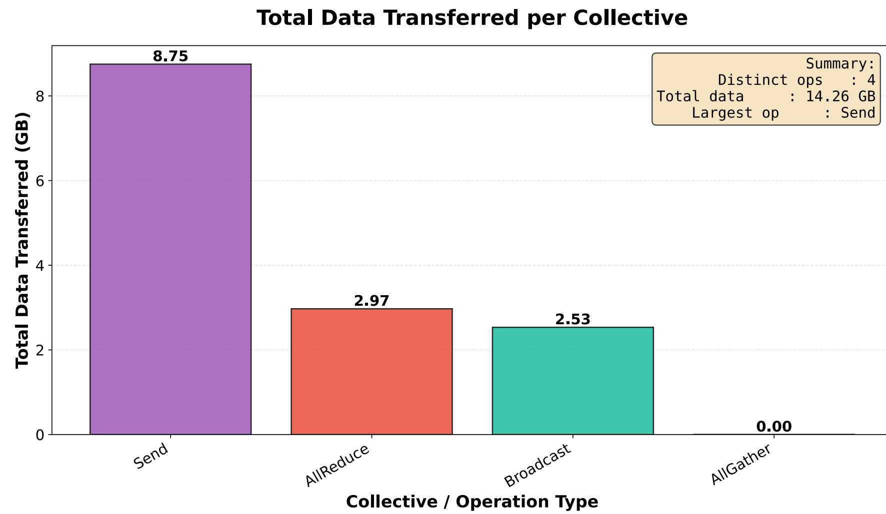


Figure 5.16: Data volume per collective type for PP=4 with virtual pipeline showing `Send` surge to 8.75 GB (61.4%, 133% increase), unchanged `AllReduce` (2.97 GB), and doubled `Broadcast` (2.53 GB). Total volume increase to 14.26 GB (78% increase) reflects the communication cost of interleaved virtual-stage scheduling.

tion overhead for reduced pipeline bubbles, achieving improved GPU utilization that precisely offsets the communication cost. The optimization represents a clear example of the computation-communication tradeoff: accepting significantly higher Communication load to maintain computational efficiency.

COMMUNICATION SIGNATURE SUMMARY Baseline PP=4 exhibits point-to-point dominated traffic with 72.4% `SendRecv` operations (1 080 per 10 iterations), a continuous temporal pattern with steady `SendRecv` activity contrasting with DP's discrete bursts.

With 2 virtual stages per physical pipeline stage, PP=4 exhibits an operation explosion with 4.1× more `SendRecvs` (4 480 vs. 1 080 total; 448 vs. 108 per iteration), `SendRecv` dominance at 91.5% of operations, up from 72.4%, and communication-computation tradeoff where substantial overhead (3.3× operations) is precisely offset by bubble reduction, maintaining similar training times despite a dramatically increased network load.

The comparison reveals that virtual pipelining is not an optimization, it achieves computational efficiency by accepting substantial communication overhead, demonstrating the fundamental tradeoff between compute utilization and network load in distributed training systems.

5.3.3 Tensor Parallelism (TP=4)

Due to Nsys profiling overhead causing NCCL timeouts with the full model configuration, TP experiments were conducted using a reduced model with 8 layers and 4 attention heads. While this limits direct quantitative comparison with DP and PP results, the communication patterns and relative behaviors remain representative of TP characteristics.

Figure 5.17 shows TP communication is `AllReduce`-dominated but with dramatically higher frequency than DP: 23 980 `AllReduces` (88.3%, 2 398 per iteration), 3 204 `Broadcasts` (11.8%, 320 per iteration), and 84 `AllGathers` (0.3%).

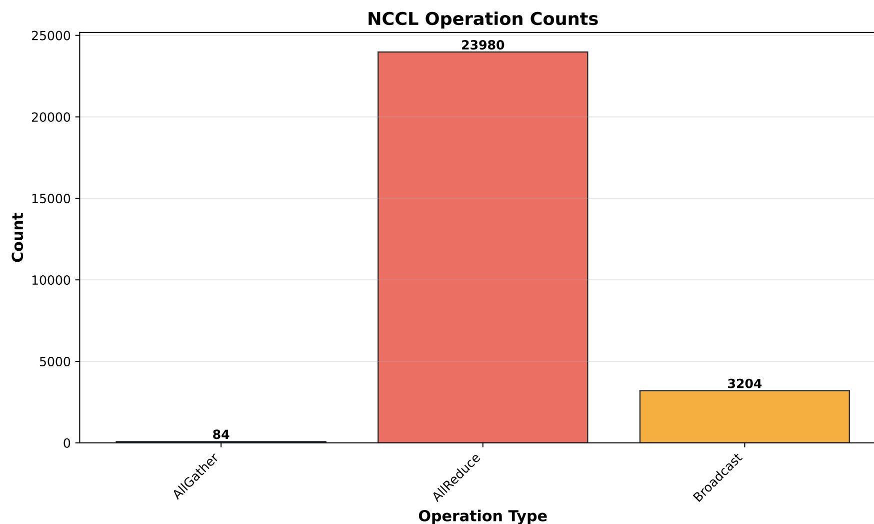


Figure 5.17: NCCL operation counts for TP=4 without SP showing `AllReduce` dominance at 88.3% with 2 398 operations per iteration, $100\times$ more frequent than DP.

The theoretical expectation from Section 2.4.2 predicts 4 `AllReduces` per layer per iteration (2 forward, 2 backward) for activation synchronization. With 8 layers this yields a baseline expectation of 32 operations per iteration; with 4 GPUs per node and 10 iterations this yields 1 280 operations in total. However, the observed 23 980 `AllReduces` per iteration represents a $\approx 18.7\times$ discrepancy from theory.

This massive operation count suggests extensive fine-grained synchronization beyond basic activation forwarding and gradient backpropagation. The high `Broadcast` frequency further confirms extremely fine-grained communication, suggesting per-sublayer or per-operation distribution of shared state across the TP group rather than coarse layer-level coordination.

This observation highlights a critical gap between theoretical TP models and implementation reality: actual transformer training involves far more communication operations than the simplified "N

operations per layer" abstraction suggests, with implications for performance prediction and optimization strategies.

The data-per-collective breakdown for TP=4 without SP (Figure 5.18) reveals extreme `AllReduce` dominance, even more pronounced than standard DP. `AllReduce` operations account for 86.02 GB (97.1% of total volume), representing the massive volume of activation synchronizations across tensor-parallel ranks at every layer and sublayer boundary. `Broadcast` operations contribute 2.57 GB (2.9%), likely for weight distribution or coordination signals while the `AllGather` again is negligible. The overwhelming `AllReduce` fraction (97.1% vs. DP's 89.8%) reflects TP's fundamentally different communication pattern: while DP synchronises once per iteration, TP synchronises at every layer operation, creating a 2,398-operations-per-iteration pattern that dwarfs DP's 30.8 operations per iteration, resulting in dramatically higher total volume (88.59 GB vs. DP's 6.37 GB) despite similar per-operation message sizes.

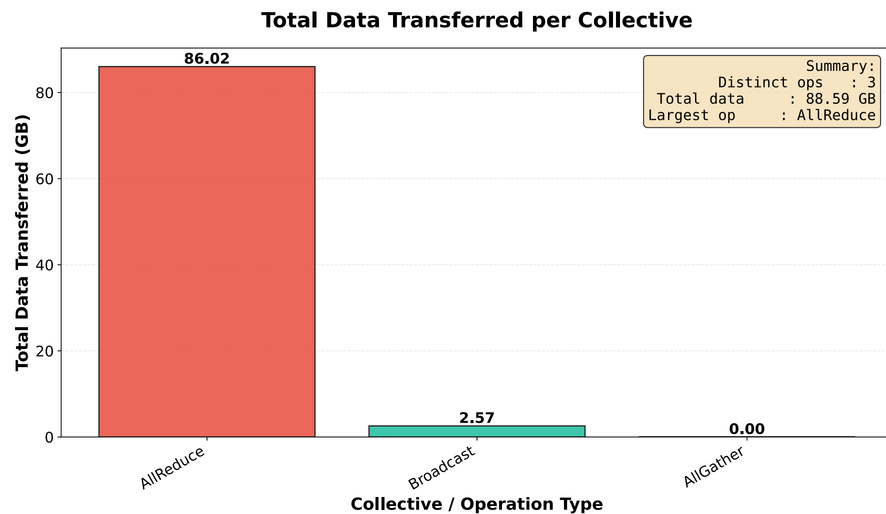


Figure 5.18: Data volume per collective type for TP=4 without SP showing extreme `AllReduce` dominance (86.02 GB, 97.1%) from high-frequency layer-boundary synchronizations. Total volume (88.59 GB) is 13.9× higher than DP's 6.37 GB, reflecting the 100× operation frequency increase. Total: 88.59 GB.

COMMUNICATION TRAFFIC COMPOSITION: TP WITH SEQUENCE PARALLELISM Enabling SP (Figure 5.19) fundamentally transforms the communication pattern: 32 724 `AllGathers` (54.8%, 3 272 per iteration), 21 120 `ReduceScatters` (35.4%, 2 112 per iteration), 3 204 `Broadcasts` (5.4%, unchanged), and only 2 900 `AllReduces` (4.9%, 290 per iteration, a 87.9% reduction).

This matches Section 2.4.4's theory where sequence partitioning requires gathering full sequences before sequence-dependent operations and scattering gradients afterward. Total operation count increases

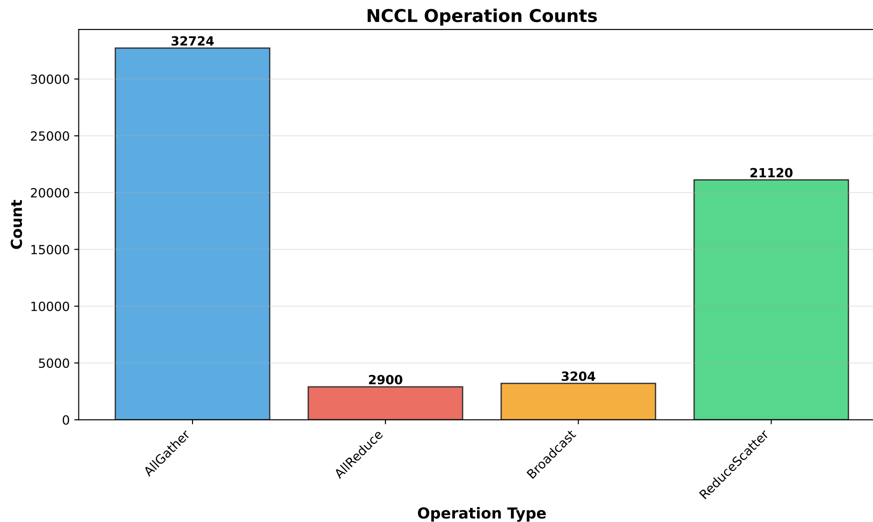


Figure 5.19: NCCL operation counts for TP=4 with SP showing a dramatic shift to `AllGather` (54.8%) and `ReduceScatter` (35.4%) dominance, with a 120% total-operation increase.

120%, from 27 268 to 59 948, reflecting the communication overhead of sequence coordination.

Enabling SP fundamentally redistributes volume across collective types (Figure 5.20), yielding lower total volume despite more operations. `AllGather` operations now dominate with 31.88 GB (54.4% of total volume), handling sequence gathering before sequence-dependent operations. `ReduceScatter` operations contribute 20.62 GB (35.2%), scattering gradients back across sequence partitions during the backward pass. `AllReduce` volume drops dramatically to 3.52 GB (6.0%), down from 86.02 GB without SP, a 95.9% reduction reflecting the shift from full-tensor synchronization to partitioned operations. `Broadcast` remains unchanged at 2.57 GB (4.4%). The total volume reduction from 88.59 GB to 58.59 GB (34% decrease) despite a 120% increase in operation count reveals SP's efficiency: by partitioning sequences and only gathering/scattering when necessary, SP reduces redundant data movement even though it requires more frequent communication.

TEMPORAL PATTERN COMPARISON The NCCL Trace Profiler timeline comparison visually confirms the pattern shift. Without SP (Figure 5.21), regular pink `AllReduce` bars appear at layer boundaries with predictable spacing. With SP (Figure 5.22), a much denser mixture of pink (`AllReduce`), blue (`ReduceScatter`), and cyan (`AllGather`) operations creates nearly continuous communication throughout forward and backward passes.

COMMUNICATION SIGNATURE SUMMARY TP without SP exhibits high-frequency collective traffic with 2 398 `AllReduces` per iteration, 100× more than DP, fine-grained synchronization at layer and sub-

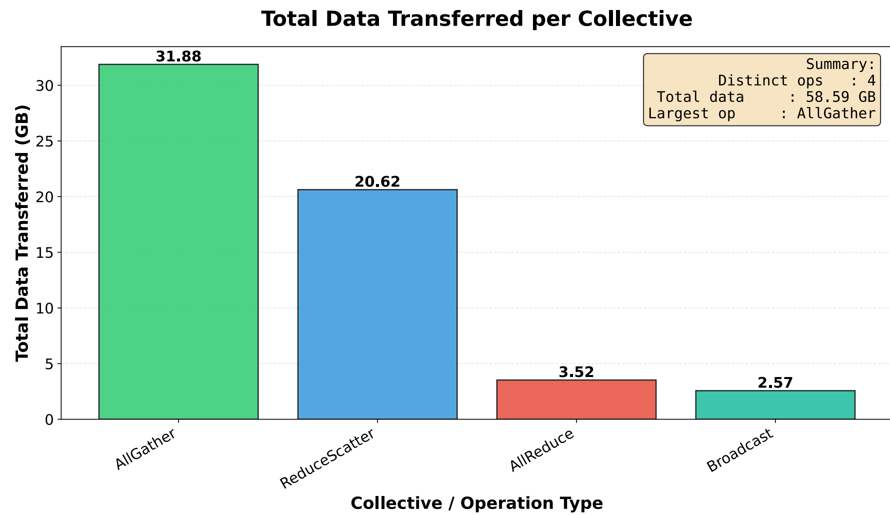


Figure 5.20: Data volume per collective type for TP=4 with SP showing `AllGather` dominance (31.88 GB, 54.4%) and `ReduceScatter` contribution (20.62 GB, 35.2%), with dramatic `AllReduce` reduction from 86.02 GB to 3.52 GB a 95.9% decrease. Total volume decreases to 58.59 GB (34% reduction) despite 120% more operations, revealing SP's efficiency through sequence partitioning.

Figure 5.21: NCCL Trace Profiler timeline for TP=4 without SP showing a regular `AllReduce` pattern at layer boundaries.

Figure 5.22: NCCL Trace Profiler timeline for TP=4 with SP showing dense heterogeneous communication with continuous mixed operation types.

layer boundaries and a regular temporal pattern aligned with forward/backward passes.

TP with SP exhibits `AllGather/ReduceScatter` dominance (90.2% combined), an operation-count explosion with a 120% increase to 5 995 operations per iteration, dense heterogeneous traffic with continuous mixed operation types.

5.3.4 Expert Parallelism (EP=4)

Figure 5.23 shows the operation counts for the AlltoAll dispatcher. `SendRecv` dominates overwhelmingly at 7 680 operations (82.5%, 768 per iteration), followed by 1 364 `AllGathers` (14.7%, 136.4 per iteration), 260 `AllReduces` (2.8%, 26 per iteration), and 4 `Broadcasts` (0.04%).

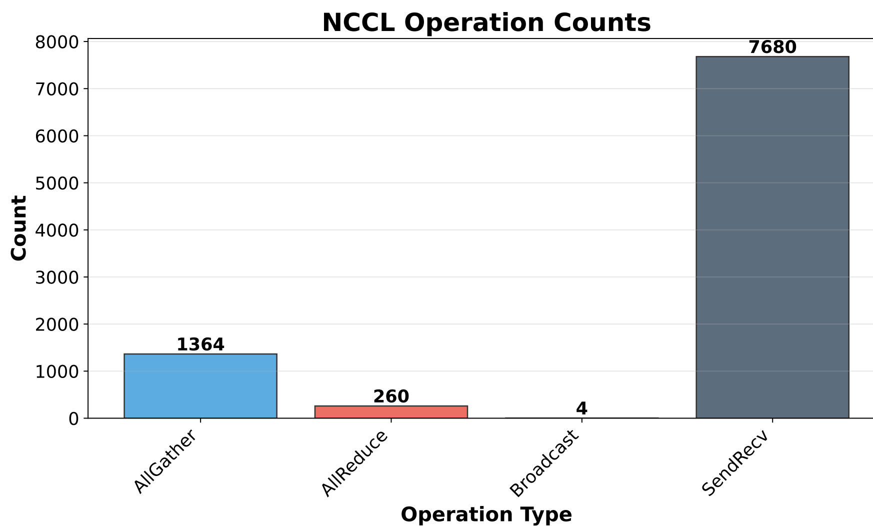


Figure 5.23: NCCL operation counts for EP=4 with AlltoAll token dispatcher showing `SendRecv` dominance (82.5%). Both A40 and A100 exhibit identical counts.

The 768 `SendRecv` operations per iteration across 4 MoE layers yield roughly 192 operations per layer. With EP=4, each rank must exchange tokens with up to 3 remote peers in both the forward and backward passes; the observed count is consistent with multiple rounds of bidirectional transfer per layer, gathering routed tokens before expert computation and returning processed results afterward. The residual 136.4 `AllGathers` per iteration likely service non-expert synchronization duties, while the 26 `AllReduces` per iteration handle gradient synchronization for non-expert parameters.

Examining the volume distribution across collective types (Figure 5.24) reveals the data-movement implications of the AlltoAll dispatcher's point-to-point strategy. `Send` operations dominate overwhelmingly with 40.04 GB (76.8% of total volume), representing the massive volume of point-to-point token transfers between expert ranks.

AllReduce operations contribute 11.43 GB (21.9%), handling gradient synchronization for non-expert parameters. Broadcast operations transfer 0.64 GB (1.2%). Notably, AllGather contributes below 10 MB.

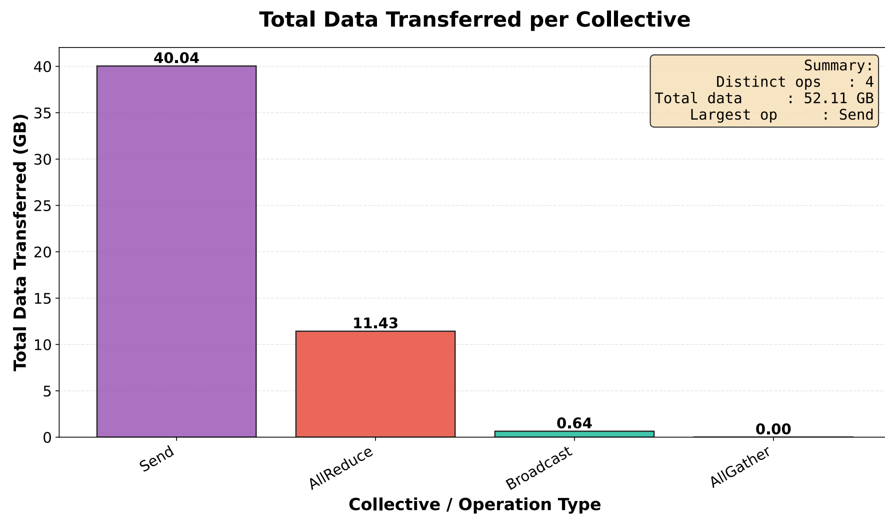


Figure 5.24: Data volume per collective type for EP=4 with AlltoAll dispatcher showing overwhelming Send dominance (40.04 GB, 76.8%), with AllReduce (11.43 GB, 21.9%) handling non-expert synchronization. The absence of ReduceScatter and near-zero AllGather volume confirms the complete replacement of collective operations with point-to-point transfers. Total volume (52.11 GB) is 62% higher than AllGather's 32.17 GB.

COMMUNICATION TRAFFIC COMPOSITION: ALLGATHER DISPATCHER

In contrast to the AlltoAll dispatcher's point-to-point approach, the AllGather dispatcher exhibits a communication pattern dominated by collective operations (Figure 5.25): 5 204 AllGathers (55.9%, 520.4 per iteration), 3 840 ReduceScatters (41.3%, 384 per iteration), 260 AllReduces (2.8%, 26 per iteration), and 4 Broadcasts (0.04%). The total operation count (9 308) is identical to the AlltoAll dispatcher, confirming that the two strategies differ in which collectives are issued rather than how many.

With 4 experts distributed across 4 ranks and 4 MoE layers (every 2nd layer of the 16-layer model), the theoretical expectation is approximately 8 token-routing operations per MoE layer per iteration. The observed 520 AllGathers per iteration suggests roughly 130 operations per MoE layer, indicating multiple token exchanges per layer, presumably for gathering tokens before expert computation and scattering results afterward.

The ReduceScatter operations, 384 per iteration, ≈96 per MoE layer, handle the reverse dataflow: distributing gradients back to their originating ranks during the backward pass. The combination of AllGather for forward token routing and ReduceScatter for backward

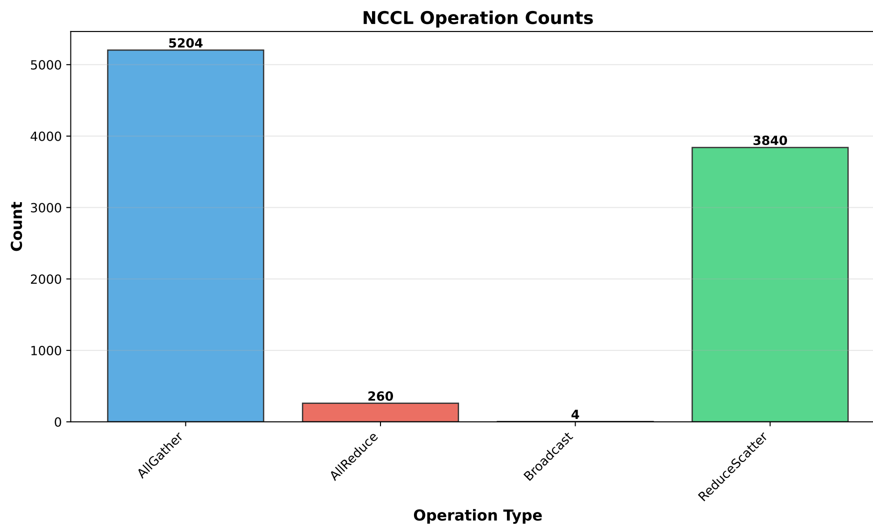


Figure 5.25: NCCL operation counts for EP=4 with AllGather token dispatcher showing `AllGather` (55.9%) and `ReduceScatter` (41.3%) dominance. Both A40 and A100 exhibit identical counts.

gradient distribution creates a symmetric communication pattern unique to the AllGather dispatcher type, contrasting with the AlltoAll dispatcher's `SendRecv`-dominated signature.

In contrast to the AlltoAll dispatcher's point-to-point volume profile, the AllGather dispatcher's data-per-collective breakdown (Figure 5.26) reveals a balanced distribution across collective operation types. `AllReduce` operations account for 11.43 GB (35.5% of total volume), handling gradient synchronization for non-expert parameters—identical to the AlltoAll dispatcher. `AllGather` operations contribute 10.06 GB (31.3%), gathering tokens before expert computation in the forward pass. `ReduceScatter` operations transfer 10.04 GB (31.2%), distributing gradients back to originating ranks during the backward pass. The near-perfect balance between `AllGather` (10.06 GB) and `ReduceScatter` (10.04 GB) confirms the symmetric forward-backward dataflow characteristic of this dispatcher type, a stark contrast to AlltoAll's 40.04 GB `Send` dominance. `Broadcast` operations contribute only 0.64 GB (2.0%). The total volume of 32.17 GB represents a 38% reduction compared to AlltoAll's 52.11 GB, demonstrating the efficiency of collective operations over point-to-point transfers for expert routing.

COMMUNICATION SIGNATURE SUMMARY EP with AlltoAll token dispatcher exhibits `SendRecv` dominance (82.5% of operations, 76.8% of volume) and a higher total communication volume, 52.11 GB vs. AllGather's 32.17 GB, a 62% increase, due to the point-to-point exchange pattern, while AllGather token dispatcher exhibits `AllGather/ReduceScatter` co-dominance at a combined 97.2%. The two dispatcher types create distinctly different signatures: the All-

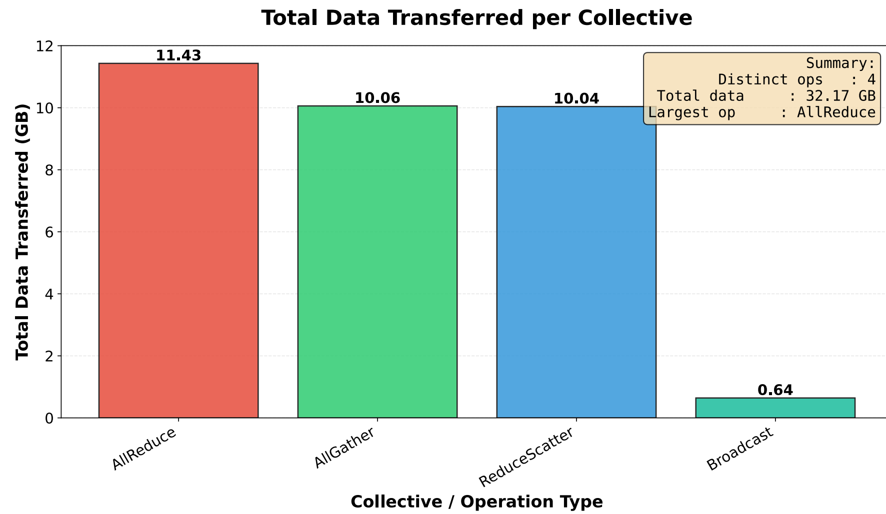


Figure 5.26: Data volume per collective type for EP=4 with AllGather dispatcher showing balanced distribution: `AllReduce` (11.43 GB, 35.5%), `AllGather` (10.06 GB, 31.3%), and `ReduceScatter` (10.04 GB, 31.2%). The near-perfect `AllGather/ReduceScatter` symmetry confirms the paired forward-backward communication pattern.

toAll dispatcher's `SendRecv` dominance with absent `ReduceScatter` contrasts sharply with the AllGather dispatcher's paired `AllGather/ReduceScatter` pattern, enabling reliable identification of the specific MoE communication strategy from traces alone.

5.3.5 Theoretical Validation of Communication Volume

The theoretical communication volume formulas presented in Sections 2.4.1-2.4.5 provide approximations of per-device network traffic based on model architecture and parallelism configuration. This section compares these theoretical predictions with the observed communication volumes measured from NCCL debug logs to validate the theoretical models and quantify the gap between idealized formulas and implementation reality.

VOLUME ACCOUNTING METHODOLOGY In Chapter 2, communication volumes are presented at varying scopes, so to enable direct comparison, all theoretical calculations and observed measurements in this section are presented on a per-rank basis for 10 training iterations. Figure 5.27 through Figure 5.31 show the observed per-rank communication volume distributions for each parallelism strategy.

For the experimental configuration, the model parameters are: $L = 16$ layers (8 for TP experiments due to profiling constraints), hidden dimension $h = 512$, sequence length $s = 1024$, vocabulary size $V = 50{,}257$, microbatch size $b = 4$, and mixed precision with $\beta = 2$ bytes

per element. The total parameter count is calculated using the formula from Section 2.4.1:

$$\begin{aligned} P &= 12Lh^2 \left(1 + \frac{1}{12h} + \frac{1}{12Lh}\frac{V+s}{h}\right) \\ &= 12 \times 16 \times 512^2 \left(1 + \frac{1}{6144} + \frac{51281}{50331648}\right) \\ &\approx 50.3 \text{ M} \end{aligned} \tag{5.2}$$

DATA PARALLELISM VOLUME VALIDATION The theoretical communication volume for DP gradient synchronization per rank per iteration, as derived in Section 2.4.1, is:

$$V_{\text{DP}} = 2 \cdot \frac{n_{\text{DP}} - 1}{n_{\text{DP}}} \cdot \frac{P}{n_{\text{TP}} \cdot n_{\text{PP}}} \cdot \beta \tag{5.3}$$

For the experimental configuration with $n_{\text{DP}} = 4$, $n_{\text{TP}} = 1$, $n_{\text{PP}} = 1$, $P = 50.4 \times 10^6$ parameters, and $\beta = 2$ bytes:

$$V_{\text{DP,iter}} = 2 \cdot \frac{3}{4} \cdot 50.4 \times 10^6 \cdot 2 = 151.2 \text{ MB/iteration}$$

For 10 iterations: $V_{\text{DP,theoretical}} = 151.2 \times 10 = 1.512$ GB per rank.

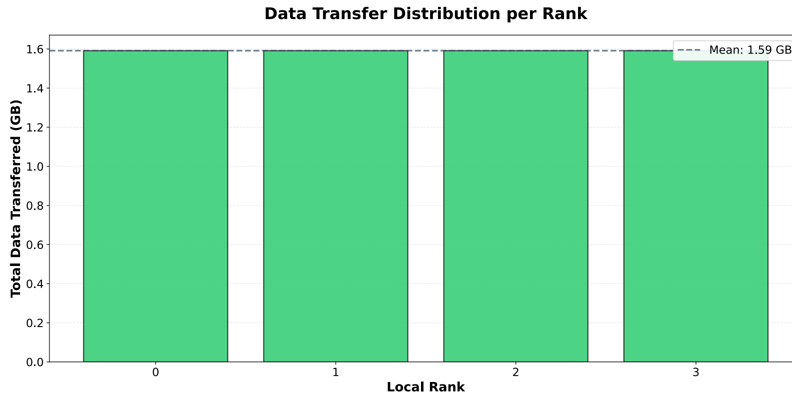


Figure 5.27: DP=4: Per-rank data transfer distribution over 10 training iterations. Mean communication volume is 1.59 GB per rank, exhibiting a uniform distribution across ranks.

The observed mean of 1.59 GB per rank (Figure 5.27) yields a ratio of 1.05×, a 5% discrepancy that validates the theoretical model's accuracy for gradient synchronization. The small excess (0.078 GB per rank) could be attributable to auxiliary `Broadcast` operations (0.64 GB total, 0.16 GB per rank) for framework initialization. The uniform distribution across all 4 ranks confirms that `AllReduce` operations distribute load symmetrically.

PIPELINE PARALLELISM VOLUME VALIDATION The theoretical volume exchanged between adjacent pipeline stages per microbatch, as established in Section 2.4.3, is:

$$V_{\text{PP}} = b \cdot s \cdot h \cdot \beta \tag{5.4}$$

With $b = 4$, $s = 1024$, $h = 512$, and $\beta = 2$ bytes:

$$V_{\text{PP,single}} = 4 \times 1024 \times 512 \times 2 = 4.19 \text{ MB/microbatch}$$

With PP=4 creating 3 stage boundaries, 16 microbatches per iteration, and bidirectional transfer (forward activations + backward gradients):

$$V_{\text{PP,iter}} = 4.19 \times 2 \times 16 = 134.1 \text{ MB/iteration per boundary}$$

On a per-rank basis, the pipeline topology creates asymmetry: edge ranks (0, 3) participate in 1 boundary each, while middle ranks (1, 2) participate in 2 boundaries each. For 10 iterations (point-to-point only):

- **Edge ranks:** $134.1 \times 1 \times 10 = 1.34$ GB per rank
- **Middle ranks:** $134.1 \times 2 \times 10 = 2.68$ GB per rank

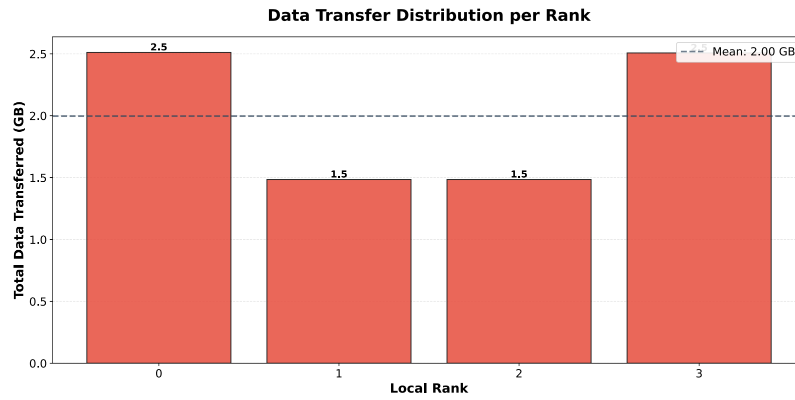


Figure 5.28: PP=4: Per-rank data transfer distribution over 10 training iterations. Mean communication volume is 2.00 GB per rank, with a pronounced asymmetric pattern where boundary stages (ranks 0 and 3) communicate more than interior stages (ranks 1 and 2).

However, the observed distribution shows the opposite pattern: edge ranks transfer 2.5 GB each while middle ranks transfer 1.5 GB each. This counterintuitive result reveals fundamental gaps in the point-to-point model:

The aggregate measurements show `AllReduce` (2.97 GB total, 0.74 GB per rank) and `Broadcast` (1.27 GB total, 0.32 GB per rank) operations entirely absent from the point-to-point formula, adding approximately 1.06 GB per rank on average. The edge ranks bear disproportionate responsibility for loss computation, optimiser coordination, batch assembly, and checkpoint management. Even accounting for global operations, edge ranks exceed expectations while middle ranks show reasonable agreement, indicating that boundary stages handle substantial coordination overhead beyond standard collectives.

MIXED VOLUME ESTIMATES A notable observation is that the theoretical model's per-rank errors are nearly complementary: the cluster-wide theoretical total ($2 \times 1.34 + 2 \times 2.68 = 8.04$ GB) closely matches the observed total ($2 \times 2.50 + 2 \times 1.50 = 8.00$ GB), despite the per-rank distribution being inverted. This suggests that the theoretical and observed volumes may simply be swapped between edge and mid ranks.

Comparing cross-wise, the theoretical edge volume of 1.34 GB closely approximates the observed mid-rank volume of 1.50 GB (ratio $1.12\times$), and the theoretical mid volume of 2.68 GB closely approximates the observed edge-rank volume of 2.50 GB (ratio $0.93\times$). Both cross-comparisons are substantially more accurate than the direct per-rank ratios of $1.87\times$ and $0.56\times$ reported in Table 5.1. This indicates that the theoretical model correctly estimates the total communication volume but misattributes it: the point-to-point formula assigns higher volume to mid ranks based on boundary count, whereas in practice edge ranks bear the higher load due to coordination responsibilities. Alternatively, Megatron-LM may distribute pipeline stages such that ranks 0 and 3 are actually interior stages rather than boundary stages, in which case the theoretical model's per-rank attribution would be correct and the apparent inversion would simply reflect a different rank-to-stage mapping than assumed.

At the cluster level, these opposing errors cancel almost exactly: the purely theoretical mean of 2.01 GB per rank deviates by less than 1% from the observed mean of 2.00 GB. For practical capacity planning, this implies that PP volume models can be applied at the cluster level with high accuracy, but per-rank estimates require accounting for the additional coordination overhead on edge ranks that the boundary-count model does not capture.

TENSOR PARALLELISM VOLUME VALIDATION The theoretical TP communication volume per device per microbatch, as derived in Section 2.4.2, is:

$$V_{\mathrm{TP}} = L_{\mathrm{stage}} \cdot 8 \cdot b \cdot s \cdot h \cdot \frac{n_{\mathrm{TP}} - 1}{n_{\mathrm{TP}}} \cdot \beta \tag{5.5}$$

For the 8-layer TP model with $L_{\mathrm{stage}} = 8$, $b = 4$, $s = 1024$, $h = 512$, $n_{\mathrm{TP}} = 4$, and $\beta = 2$:

$$V_{\mathrm{TP,microbatch}} = 8 \times 8 \times 4 \times 1024 \times 512 \times \frac{3}{4} \times 2 = 201.3\ \mathrm{MB/microbatch}$$

With 4 microbatches per iteration and 10 iterations: $V_{\mathrm{TP,theoretical}} = 201.3 \times 4 \times 10 = 8.05$ GB per rank.

The observed mean of 22.15 GB per rank yields a ratio of $2.75\times$, indicating that the theoretical model underestimates actual communication. This discrepancy stems from the implementation performing synchronization at a much finer granularity than the theoretical model

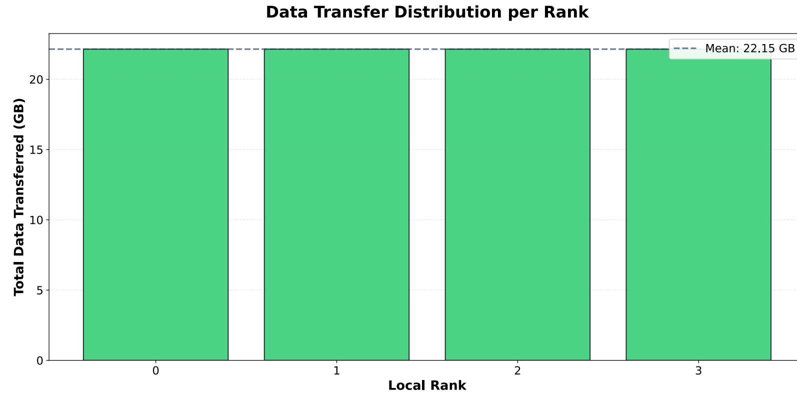


Figure 5.29: TP=4: Per-rank data transfer distribution over 10 training iterations. Mean communication volume is 22.15 GB per rank, showing a uniform distribution across ranks.

assumes. The theoretical model expects 4 `AllReduce` operations per layer (2 forward, 2 backward), yielding $8 \times 4 = 32$ operations per rank per iteration. The aggregate measurements, however, show 23,980 total `AllReduce` operations (599.5 per rank per iteration), an $18.7\times$ underestimation of operation count. This indicates that Megatron-LM decomposes each theoretical per-layer collective into many smaller synchronization points, for instance synchronizing individual sublayer components such as attention projections, FFN layers, and normalization statistics separately.

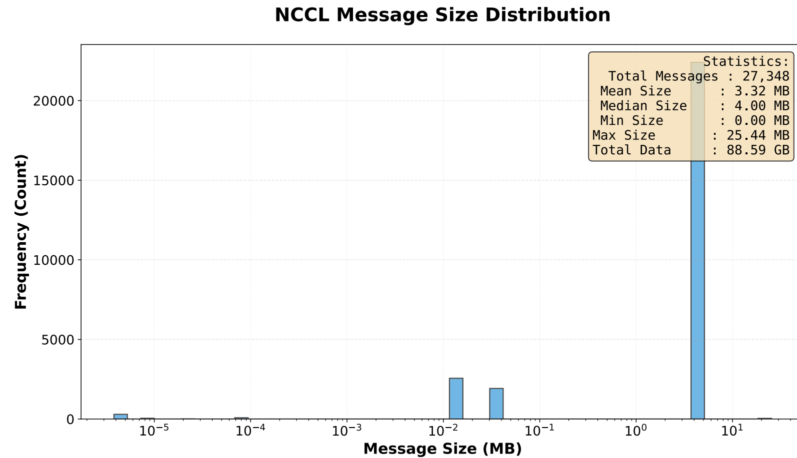


Figure 5.30: TP=4: Message-size distribution across all ranks over 10 training iterations. The dominant peak at ≈4 MB with over 20,000 messages corresponds to the theoretical $b \times s \times h$ activation tensor size, while secondary clusters at 10^{-2}–10^{-1} MB represent coordination and control messages.

The message-size distribution in Figure 5.30 confirms that the theoretical per-operation size assumption is correct: the expected activation tensor size is $b \times s \times h \times \beta = 4 \times 1024 \times 512 \times 2 = 4{,}194{,}304$

bytes = 4.0 MB, which matches the dominant peak and the observed median message size of 4.00 MB exactly. Crucially, this dominant peak accounts for the vast majority of all 27,348 messages, with over 20,000 operations concentrated at the theoretical tensor size. The volume discrepancy of $2.75\times$ is therefore driven almost entirely by the $18.7\times$ operation count surplus at the expected message size: Megatron-LM performs far more ≈ 4 MB collectives per layer than the theoretical 4, indicating that the implementation decomposes each layer into many independently synchronized sublayer components, each communicating the full activation tensor. The secondary clusters at 10^{-2}–10^{-1} MB visible in Figure 5.30 represent a comparatively small number of coordination and control messages that contribute negligibly to the total volume. The uniform per-rank distribution confirms that TP's matrix partitioning creates perfectly balanced workload despite this fine-grained decomposition.

EXPERT PARALLELISM VOLUME VALIDATION The theoretical communication volume for EP with AlltoAll-based token dispatch, as established in Section 2.4.5, provides total cluster volume:

$$V_{\text{EP,cluster}} \approx 4 \cdot B \cdot S \cdot k \cdot h \cdot \left(1 - \frac{1}{n_{\text{EP}}}\right) \cdot \beta \quad (5.6)$$

For the MoE configuration with $B = 64$, $S = 1024$, top-$k = 1$, $h = 512$, $n_{\text{EP}} = 4$, and $\beta = 2$:

$$V_{\text{EP,cluster,layer}} = 4 \times 64 \times 1024 \times 1 \times 512 \times \frac{3}{4} \times 2 = 201.3 \text{ MB/layer/iteration}$$

With 4 MoE layers and converting to per-rank: $V_{\text{EP,per-rank}} = \frac{201.3 \times 4}{4} = 201.3$ MB/iteration for MoE communication only.

Adding non-expert `AllReduce` for the 12 non-MoE layers, proportional to DP: $V_{\text{non-expert}} \approx 151.2 \times \frac{12}{16} = 113.4$ MB/iteration.

Total theoretical per-rank for 10 iterations: $(201.3 + 113.4) \times 10 = 3.15$ GB per rank.

The observed mean of 13.03 GB per rank yields a ratio of $4.1\times$, indicating substantial implementation overhead. This discrepancy possibly arises from the $O(n_{\text{EP}}^2)$ point-to-point communication pattern generating 16 transfer pairs even when many are small; multiple routing rounds per MoE layer for gating logits, token routing, and gradient distribution; capacity factor padding inflating routing buffers and the 11.43 GB of observed `AllReduce` volume, at 2.86 GB per rank, for non-expert synchronization exceeding the theoretical estimate. The uniform per-rank distribution demonstrates that despite using point-to-point `Send` operations, token routing remains balanced across expert ranks.

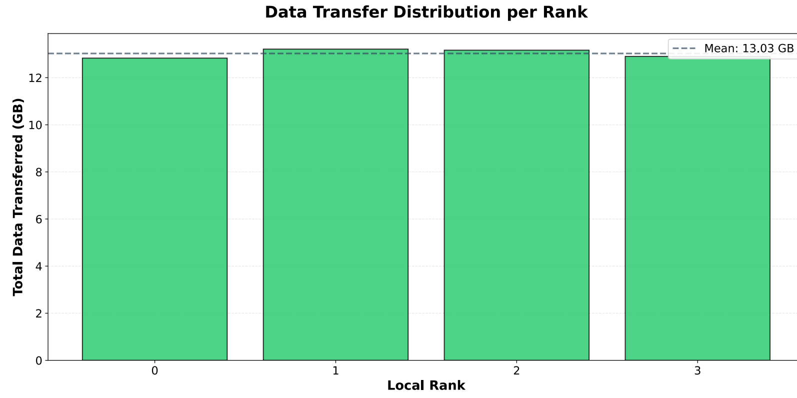


Figure 5.31: EP=4 (AlltoAll): Per-rank data transfer distribution over 10 training iterations. Mean communication volume is 13.03 GB per rank.

VALIDATION SUMMARY Table 5.1 summarizes the theoretical predictions and observed volumes across all parallelism strategies.

Table 5.1: Comparison of theoretical and observed communication volumes per rank.

Strategy	Theor. (GB)	Obs. (GB)	Ratio	Primary Discrepancy
DP=4	1.51	1.59	1.05×	Broadcast overhead
PP=4 (edge)	1.34	2.50	1.87×	Missing global ops
PP=4 (mid)	2.68	1.50	0.56×	Incomplete topology model
TP=4	8.05	22.15	2.75×	Fine-grained sublayer sync
EP=4	3.15	13.03	4.10×	Capacity factor

The validation reveals that theoretical formulas serve different roles depending on parallelism strategy. Data Parallelism (DP) achieves near-perfect agreement (1.05×), validating that gradient synchronization dominates and the formula accurately captures per-rank communication. Pipeline Parallelism (PP) exhibits structural incompleteness: the point-to-point formula omits global operations and coordination overhead, and fails to predict the observed asymmetry where edge ranks communicate more than middle ranks due to additional responsibilities. Notably, cross-comparing theoretical edge volumes against observed mid volumes and vice versa, yields ratios of 1.12× and 0.93× respectively, indicating that the model correctly predicts total volume but may misattribute it between ranks. Tensor Parallelism (TP) underestimates by 2.75× due to the implementation decomposing each layer into far more independently synchronized sublayer components than the theoretical 4-per-layer model predicts, producing 18.7× more operations at the expected activation tensor size, as confirmed by the message-size distribution; the uniform per-rank distribution

confirms balanced partitioning. Expert Parallelism (EP) shows the largest discrepancy (4.1×) from $O(n^2)$ communication patterns, capacity factor overhead, and multi-round routing not captured in the idealized formula.

These results establish that while the theoretical formulas from Chapter 2 capture the fundamental scaling behavior, practical implementations introduce systematic overheads that must be empirically calibrated. The NCCL tracing methodology accurately measures these real-world volumes, bridging the gap between theoretical models and production-system behavior. For performance modeling, DP formulas can be used directly, PP formulas should be applied at the cluster level where edge and mid rank errors cancel, while TP requires ≈3× and EP requires ≈4× empirical correction factors specific to Megatron-LM's implementation.

6 DISCUSSION AND OUTLOOK

This thesis presents the NCCL Trace Profiler, a tool for fine-grained analysis of communication patterns in distributed LLM training. The profiling methodology addresses a critical gap by correlating NCCL debug logs with Nsys kernel traces using sequence alignment techniques, combining semantic metadata with precise timing information without requiring source code modifications.

The experimental evaluation in Chapter 5 demonstrates the profiler's effectiveness across four parallelism strategies on two hardware architectures. Several key findings emerge from the communication pattern analysis.

COMMUNICATION SIGNATURE CHARACTERISATION The profiler successfully identifies distinct communication signatures for each parallelism strategy, enabling reliable strategy identification from traces alone. DP exhibits `AllReduce` dominance (71.4% of operations), PP shows `SendRecv` dominance (72%), TP maintains high-frequency `AllReduce/ReduceScatter` patterns (88% combined), and EP demonstrates either `AllGather/ReduceScatter` co-dominance (97.2%) or `SendRecv` dominance (82.5%) depending on the token dispatcher implementation. These signatures are architecture-invariant: the A40 and A100 nodes exhibit identical operation counts and distributions, confirming that communication patterns are determined by parallelism strategy rather than interconnect technology.

THEORETICAL MODEL VALIDATION The comparison of theoretical communication volume formulas against observed measurements reveals strategy-dependent accuracy. DP achieves excellent agreement with a ratio of 1.05× between observed and theoretical volumes, validating that gradient synchronization formulas accurately capture per-rank communication. TP underestimates by 2.75×, with 18.7× more operations observed than theoretical predictions suggest. The message-size distribution confirms that the per-operation size assumption of $b \times s \times h$ is correct, with the vast majority of messages concentrated at the expected tensor size, the volume discrepancy is therefore driven by Megatron-LM decomposing each layer into many more independently synchronized sublayer components than the idealized 4-per-layer model assumes, each communicating full activation-sized tensors. EP shows the largest discrepancy at 4.1×, attributable to $O(n^2)$

communication patterns, capacity factor overhead, and multi-round routing not captured in theoretical formulas. PP exhibits structural incompleteness in its theoretical model: the point-to-point formula omits global operations and coordination overhead, and fails to predict the observed asymmetry where boundary stages communicate more than interior stages. However, cross-comparing theoretical edge volumes against observed mid volumes and theoretical mid volumes against observed edge volumes reveals that the model correctly predicts total volume but may misattribute it between rank types, with per-rank errors cancelling almost exactly at the cluster level at less than 1% deviation. The theoretical model assigns higher volume to mid ranks based on boundary count, whereas in practice edge ranks bear the higher load due to coordination responsibilities.

These results establish that while the theoretical formulas from Chapter 2 capture fundamental scaling behavior, practical implementations introduce systematic overheads requiring empirical calibration. For performance modeling, DP formulas can be applied directly, PP formulas are accurate at the cluster level but require full per-rank calibration to capture the edge-versus-mid asymmetry, while TP requires approximately $3\times$ and EP approximately $4\times$ empirical correction factors specific to Megatron-LM's implementation.

6.1 CURRENT LIMITATIONS

Despite its effectiveness, the current implementation of the NCCL Trace Profiler has several important limitations that constrain its applicability and performance.

ALIGNMENT ACCURACY AND UNMATCHED OPERATIONS The Needleman-Wunsch alignment algorithm, while achieving an average F1 score of 89.3% across validation scenarios, does not guarantee 100% operation matching. Several factors contribute to unmatched operations: NCCL's internal operation fusion (e.g., combining `Send`/`Recv` pairs into `SendRecv` kernels) and the fundamental mismatch between CPU-side log entries and GPU-side kernel execution granularity as detailed in Section 4.1.

For the communication volume analysis presented in Chapter 5, measurements were therefore derived directly from NCCL debug logs to ensure completeness, as the profiler's matched operations represent only a subset of total communication. The profiler's per-operation bandwidth calculations remain accurate for individual matched operations but would underestimate aggregate throughput due to unmatched events.

A more robust solution would be to be able to export the NCCL NVTX annotations, which contain complete semantic metadata, di-

rectly to the SQLite format alongside kernel timing data. As discussed in Section 3.4, these annotations exist within the raw `.nsys-rep` trace format but are deliberately excluded during the SQLite export process. Native support for exporting NCCL NVTX markers with their associated metadata would eliminate the need for sequence alignment entirely, enabling exact operation-to-kernel correspondence and fully accurate bandwidth measurements.

PROFILING SCOPE Another significant limitation stems from the requirement to profile entire training runs rather than individual iterations. This constraint arises from the nature of communication patterns in LLM training and the global alignment approach employed by the Needleman-Wunsch algorithm.

As demonstrated in Chapter 5, LLM training exhibits highly repetitive communication patterns across iterations. The Needleman-Wunsch algorithm exploits this repetition to achieve robust alignment even when individual operations are missing from the logs. However, profiling entire training runs produces substantial data volumes. For runs with thousands of iterations, the Nsys SQLite database can grow to multiple gigabytes, while NCCL debug logs can reach hundreds of megabytes, particularly for configurations with high operation counts (e.g., TP=8 with SP). This makes the profiling workflow cumbersome for long-running production jobs, and Nsys itself imposes limitations on trace duration due to buffer constraints. A critical enhancement for future work would therefore be enabling single-iteration profiling with accurate alignment. The primary obstacle is establishing a reliable synchronization point between the two data sources.

For full training run profiling, the Needleman-Wunsch algorithm compensates by aligning complete operation sequences. However, restricting analysis to a single iteration eliminates this global context. Additionally, long sequences of identical operations (e.g., 100 consecutive `AllReduce` operations with identical message sizes) can produce multiple valid alignments with equivalent scores, making single-iteration alignment particularly challenging.

A potential approach would involve inserting explicit NCCL barrier operations (e.g., `ncclAllReduce` with zero bytes) at iteration boundaries. These barriers would appear in both the debug logs and the Nsys trace, serving as unambiguous synchronization points. However, the barriers would represent artificial communication operations not present in unmodified training, potentially complicating the interpretation of communication patterns, necessitating filtering.

PERFORMANCE The current implementation is strictly single-threaded, processing each rank's trace sequentially. This creates performance bottlenecks when analyzing multi-rank training runs. However, the profiler's architecture is inherently amenable to parallelization, as

each rank's trace can be processed completely independently. Using Python's `multiprocessing` module, the profiler could spawn worker processes to handle individual ranks concurrently, potentially reducing runtime from minutes to seconds on modern multi-core workstations.

6.2 FUTURE DIRECTIONS

Beyond addressing the identified limitations, several promising research directions could extend the capabilities and impact of this work.

AUTOMATIC PERFORMANCE ANOMALY DETECTION The profiler currently provides raw trace data and statistics, leaving pattern recognition and anomaly identification to human analysts. Machine learning techniques could identify outlier operations, detect communication imbalances across ranks, or recognize deviations from expected patterns without requiring deep distributed systems expertise.

CROSS-RUN COMPARATIVE ANALYSIS The current profiler analyzes individual training runs in isolation. Developing tools to compare communication patterns across multiple runs would enable configuration comparison between different parallelism strategies, hardware comparison, framework comparison and regression detection when software updates introduce communication inefficiencies.

6.3 CONCLUSION

The NCCL Trace Profiler represents a significant advance in distributed machine learning observability, providing the first tool capable of fine-grained, per-operation analysis of collective communication patterns without requiring source code modification. The experimental evaluation demonstrates its effectiveness: the profiler successfully characterises distinct communication signatures for each parallelism strategy.

The theoretical validation reveals important insights for practitioners: DP communication formulas can be applied directly with excellent accuracy (1.05× ratio), PP formulas achieve near-exact cluster-level totals despite inverting the per-rank distribution between edge and mid ranks, while TP and EP require empirical correction factors of approximately 3× and 4× respectively to account for fine-grained synchronization and routing overhead not captured in idealized models. These findings bridge the gap between theoretical understanding and production-system behavior, providing quantitative guidance for performance modeling and capacity planning.

While current limitations around alignment accuracy, full training run requirements, and single-threaded performance limit applicability for the most demanding use cases, the proposed enhancements, particularly native NVTX marker export support, offer clear paths to addressing these constraints.

As LLMs continue to grow in scale and complexity, the need for sophisticated profiling tools will only intensify. This thesis provides both a practical tool for immediate use and a methodological foundation for future advances in understanding and optimizing distributed training workloads.

ACRONYMS

AI Artificial Intelligence. 1, 2, 8

ASIC Application-Specific Integrated Circuit. 7

CP Context Parallelism. 19, 21, 22

CPU Central Processing Unit. 7, 8, 42, 44, 45, 58–60, 92

DP Data Parallelism. v, vi, 1, 3, 13–15, 18, 21, 22, 48, 49, 52, 65–71, 74–77, 83, 87–89, 91, 92, 94

EP Expert Parallelism. v, vi, 3, 19–22, 64, 65, 79–82, 87–89, 91, 92, 94

FFN Feed-Forward Network. 6, 15, 16, 19, 86

FSDP Fully Sharded Data Parallel. 14

GB Gigabyte. x, 6–8, 12, 16, 59–63, 67–74, 76–88

Gbps Gigabit per Second. 8

GPT Generative Pre-trained Transformer. 5, 6, 63

GPU Graphics Processing Unit. v, vi, 1, 5–9, 12, 17, 20, 29, 31–33, 38, 40–42, 44–46, 48, 51, 52, 58–60, 62–65, 75, 92

HBM High-Bandwidth Memory. 7

HPC High-Performance Computing. 8

LL Low Latency. 11

LLM Large Language Model. v, vi, 1–3, 5–8, 12, 29, 30, 34, 63, 65, 91, 93, 95

MB Megabyte. 11, 61, 64, 67, 71, 80, 83–87

MoE Mixture of Experts. 6, 19–21, 63, 64, 79, 80, 82, 87

NCCL NVIDIA Collective Communications Library. v, vi, 1–5, 9, 11, 12, 23, 24, 26, 30–34, 37–48, 50, 52–57, 59–62, 64–68, 70–73, 75, 77, 79, 81, 82, 89, 91–94

NIC Network Interface Controller. 45, 60

Nsys NVIDIA Nsight Systems. v, vi, 2, 3, 23, 24, 26, 31, 32, 34, 37, 40–42, 47, 48, 53–56, 63, 75, 91, 93

NUMA Non-Uniform Memory Access. 45, 58–60

NVMe Non-Volatile Memory Express. 8

NVTX NVIDIA Tools Extension. 32, 37, 46, 52, 92, 93, 95

PCIe Peripheral Component Interconnect Express. 8, 11, 12, 45, 51, 52, 58–62

PP Pipeline Parallelism. v, vi, 1, 3, 14, 16–19, 21, 22, 38, 39, 48, 52, 64, 65, 70–75, 84, 85, 88, 89, 91, 92, 94

RDMA Remote Direct Memory Access. 8, 9

SP Sequence Parallelism. 19, 21, 64, 65, 75–79, 93

TP Tensor Parallelism. v, vi, 1, 3, 14–16, 18, 20–22, 30, 38–40, 52, 58, 65, 75–79, 82, 85–89, 91–94

TPU Tensor Processing Unit. 7

ZeRO Zero Redundancy Optimizer. 14

a | APPENDIX

A.1 DETAILED NEEDLEMAN WUNSCH EXAMPLE

To illustrate the algorithm concretely, consider aligning two short DNA sequences:

- **Sequence A**: GCAT (length $n = 4$)
- **Sequence B**: GGAT (length $m = 4$)

Scoring Parameters:

- Match score: $S_{match} = +1$
- Mismatch penalty: $S_{mismatch} = -1$
- Gap penalty: $d = -1$

Thus, the substitution matrix is:

$$S(x, y) = \begin{cases} +1 & \text{if } x = y \\ -1 & \text{if } x \neq y \end{cases}$$

A.1.1 Step-by-Step Matrix Construction

We construct a 5×5 matrix (accounting for the empty prefix row and column).

Initialization:

	ϵ	G	G	A	T
ϵ	0	-1	-2	-3	-4
G	-1				
C	-2				
A	-3				
T	-4				

Table a.1: Initial state after boundary initialization

A.1.2 Filling the Matrix

Let us calculate the cell $(1, 1)$ comparing 'G' (Seq A) and 'G' (Seq B):

- **Diagonal:** $F(0,0) + S(\text{G},\text{G}) = 0 + 1 = 1$
- **Vertical (Up):** $F(0,1) + d = -1 + (-1) = -2$
- **Horizontal (Left):** $F(1,0) + d = -1 + (-1) = -2$

$\max(1, -2, -2) = 1$. The value of cell $(1,1)$ is 1.

Repeating this process for the entire grid yields the following matrix. The optimal path is highlighted with cell shading.

	ϵ	G	G	A	T
ϵ	**0**	-1	-2	-3	-4
G	-1	**1**	0	-1	-2
C	-2	0	**0**	-1	-2
A	-3	-1	-1	**1**	0
T	-4	-2	-2	0	**2**

Table a.2: The filled Needleman-Wunsch matrix with optimal alignment path highlighted

A.1.3 Traceback and Result

Starting from the bottom-right corner (Score 2):

1. $F(4,4) = 2$. Came from Diagonal $F(3,3) = 1$ (Match T-T).
2. $F(3,3) = 1$. Came from Diagonal $F(2,2) = 0$ (Match A-A).
3. $F(2,2) = 0$. Came from Diagonal $F(1,1) = 1$ (Mismatch C-G).
4. $F(1,1) = 1$. Came from Diagonal $F(0,0) = 0$ (Match G-G).

The resulting alignment is:

```
Seq A:  G  C  A  T
        |  x  |  |
Seq B:  G  G  A  T
```

(Where | denotes a match and × denotes a mismatch)

Alignment Score Verification:

$$\begin{aligned}\text{Score} &= S(\text{G},\text{G}) + S(\text{C},\text{G}) + S(\text{A},\text{A}) + S(\text{T},\text{T}) \\ &= (+1) + (-1) + (+1) + (+1) \\ &= 2\end{aligned}$$

This matches the optimal score of $F(4,4) = 2$ found in the matrix. The alignment successfully identifies three matches (G-G, A-A, T-T) and one mismatch (C-G), with no gaps needed.

A.2 ENVIRONMENT SETUP SCRIPT

```
1  #!/bin/bash
2
3  # This script automates the setup of a conda environment for training
       Megatron models.
4  # It clones the required repositories, installs dependencies, and builds
       CUDA extensions.
5
6  #SBATCH --partition=gpu-single
7  #SBATCH --job-name="setup_env"
8  #SBATCH --nodes=1
9  #SBATCH --ntasks-per-node=64
10 #SBATCH --gres=gpu:H200:2
11 #SBATCH --mem=64G
12 #SBATCH --cpus-per-task=1
13 #SBATCH --time=4:00:00
14 #SBATCH --output="slurm/job_%x_%j.txt"
15
16 # --- Configuration ---
17 CONDA_ENV_NAME="megatron-h200"
18 MEGATRON_ROOT_PATH="/home/hd/hd_hd/hd_gn324/thesis/Megatron"
19 MEGATRON_COMMIT=73a28a1078a8da8e6062199f7f1079a52173ab77  # core_r0.13.0
20 APEX_COMMIT=bfb500c87e57f1b53fd438a6dc1504a5fce85462
21
22 export PYTHONNOUSERSITE=1
23
24 # --- Module Loading ---
25 module purge
26 module load devel/miniforge
27 module load devel/cuda/12.8
28
29 # --- 1. Repository Setup ---
30 cd $MEGATRON_ROOT_PATH || exit 1
31
32 # Clone and checkout Megatron-LM
33 if [ ! -d "megatron" ]; then
34     echo "Cloning Megatron-LM repository..."
35     git clone https://github.com/NVIDIA/Megatron-LM.git
36     mv Megatron-LM megatron
37 fi
38 cd megatron && git checkout $MEGATRON_COMMIT && cd ..
39
40 # Clone and checkout Apex
41 if [ ! -d "apex" ]; then
42     echo "Cloning Apex repository..."
43     git clone https://github.com/NVIDIA/apex
44     APEX_CLONED_MANUALLY=TRUE
45 fi
46 cd apex && git checkout $APEX_COMMIT && cd ..
47
48 # --- 2. Environment Creation ---
49 conda env remove -n $CONDA_ENV_NAME -y
50 if [[ -d "$HOME/.conda/envs/$CONDA_ENV_NAME" ]]; then
51     rm -rf "$HOME/.conda/envs/$CONDA_ENV_NAME"
52 fi
53
```

```
54  conda create -n $CONDA_ENV_NAME python=3.12 -y
55  eval "$(command conda 'shell.bash' 'hook' 2> /dev/null)"
56  conda activate $CONDA_ENV_NAME
57
58  # --- 3. Dependency Installation ---
59  conda install \
60      "gcc_linux-64==12.2.0" \
61      "gxx_linux-64==12.2.0" \
62      "rdma-core==58.0" \
63      "cuda-cudart==12.8.90" \
64      "cuda-nvrtc==12.8.93" \
65      "cuda-version==12.8" \
66      "cudnn==9.10.1.4" \
67      "cusparselt==0.7.1.0" \
68      "libcublas==12.8.4.1" \
69      "libcudnn==9.10.1.4" \
70      "libcusparse==12.5.8.93" \
71      "nccl==2.28.9.1" \
72      -c conda-forge -n $CONDA_ENV_NAME -y
73
74  pip install \
75      "torch==2.7.1" \
76      "torchaudio==2.7.1" \
77      "torchvision==0.22.1" \
78      --index-url https://download.pytorch.org/whl/cu128
79
80  pip install \
81      "tqdm==4.67.1" \
82      "requests==2.32.4" \
83      "cffi==1.17.1" \
84      "platformdirs==4.3.8" \
85      "cryptography==45.0.5" \
86      "tensorboard==2.20.0" \
87      "transformers==4.38.0" \
88      "datasets==4.0.0" \
89      "ninja==1.11.1.4" \
90      "nltk==3.9.2" \
91      "einops==0.8.1" \
92      "wandb==0.21.0" \
93      "numpy==1.26.4" \
94      "packaging==25.0" \
95      "psutil==7.0.0" \
96      "pydantic==2.11.7" \
97      "pynvml==12.0.0" \
98      "typing_extensions" \
99      "mpmath"
100
101 export FLASH_ATTENTION_FORCE_BUILD=TRUE
102 export MAX_JOBS=4
103 pip install flash-attn==2.7.4.post1 --no-build-isolation --no-cache-dir
104
105 # --- 4. Apex Installation ---
106 export NVCC_APPEND_FLAGS="--threads 4"
107 export MAX_JOBS=4
108 export APEX_PARALLEL_BUILD=8
109 export TORCH_CUDA_ARCH_LIST="9.0"
110 export APEX_CPP_EXT=1
```

```
111 export APEX_CUDA_EXT=1
112 export CC=$CONDA_PREFIX/bin/x86_64-conda-linux-gnu-gcc
113 export CXX=$CONDA_PREFIX/bin/x86_64-conda-linux-gnu-g++
114
115 cd apex
116 pip install -v --no-build-isolation .
117 cd ..
118
119 if [[ "$APEX_CLONED_MANUALLY" == "TRUE" ]]; then
120     rm -rf apex
121 fi
122
123 # --- 5. Additional Packages ---
124 conda deactivate
125 conda activate $CONDA_ENV_NAME
126
127 python -c "
128 import nltk
129 try:
130     nltk.data.find('tokenizers/punkt')
131 except LookupError:
132     nltk.download('punkt')
133 "
134
135 export MAX_JOBS=1
136 export NVTE_FRAMEWORK=pytorch
137 pip install --no-cache-dir --no-build-isolation "transformer-engine[
        pytorch]==2.5.0"
138
139 cd megatron
140 pip install -e .
141 cd ..
142
143 # --- 6. Verification ---
144 echo "Environment setup complete."
145 conda list
146 echo "---"
147 pip list
148
149 echo "Finished setting up the environment for Megatron training."
```

Listing a.1: Environment Setup Script

A.3 TRAINING SCRIPT

```
1 #!/bin/bash
2
3 # Slurm sbatch script for training GPT models using Megatron-LM with
      torchrun.
4 # Supports both dense and Mixture-of-Experts (MoE) architectures with
      profiling capabilities.
5
6 #SBATCH --job-name=megatron-train
7 #SBATCH --time=01:00:00
```

```
8   #SBATCH --output=slurm/%x-%j.txt
9   #SBATCH --error=slurm/%x-%j.err
10  #SBATCH --partition=gpu-single
11  #SBATCH --nodes=1
12  #SBATCH --ntasks-per-node=1
13  #SBATCH --cpus-per-task=16
14  #SBATCH --gres=gpu:A40:4
15  #SBATCH --mem-per-gpu=48G
16  #SBATCH --gpu-bind=closest
17  #SBATCH --mem-bind=local
18  #SBATCH --distribution=cyclic:cyclic
19  #SBATCH --hint=memory_bound
20  #SBATCH --exclusive
21  #SBATCH --export=ALL
22
23  set -e
24
25  #===================================================
26  # INPUT ARGUMENTS
27  #===================================================
28
29  MODEL_SIZE=${1:-"dense_moe"}
30  TP=${2:-1}
31  PP=${3:-1}
32  HW_NAME=${4:-"A40"}
33  CONDA_ENV_NAME=${5:-"megatron-a40"}
34  EP=${6:-1}
35  CP=${7:-1}
36  SP=${8:-"auto"}
37  OPT_FLAGS=${9:-""}
38
39  #===================================================
40  # USER CONFIGURATION
41  #===================================================
42
43  ITERATIONS=10
44  GLOBAL_BATCH_SIZE=64
45  REPRODUCIBILITY_SEED=42
46
47  MEGATRON_ROOT_PATH="/home/hd/hd_hd/hd_gn324/thesis/Megatron"
48  DATA_PATH_PREFIX="$MEGATRON_ROOT_PATH/training/data/fineweb"
49
50  PROFILE=1
51  DEBUG_MODE=0
52
53  #===================================================
54  # 1. HARDWARE & ENVIRONMENT INSPECTION
55  #===================================================
56
57  print_hardware_info() {
58      echo "=================================================="
59      echo " HARDWARE & JOB INFORMATION"
60      echo "=================================================="
61      echo "Job ID:         $SLURM_JOB_ID"
62      echo "Node List:      $SLURM_JOB_NODELIST"
63      echo "Date:           $(date)"
64      echo "--------------------------------------------------"
```

```
65      echo "CPU Info:"
66      lscpu | grep -E "Model name|Socket|Thread|NUMA|CPU\(s\)"
67      echo "----------------------------------------------------"
68      echo "Memory Info:"
69      free -h
70      echo "----------------------------------------------------"
71      echo "GPU Info (nvidia-smi):"
72      nvidia-smi --query-gpu=timestamp,name,pci.bus_id,driver_version,
            memory.total,memory.free --format=csv
73      echo "----------------------------------------------------"
74      echo "NVLink Topology:"
75      nvidia-smi topo -m
76      echo "===================================================="
77  }
78
79  print_hardware_info
80
81  #====================================================
82  # 2. ENVIRONMENT SETUP
83  #====================================================
84
85  echo "--- Loading Modules and Conda Environment ---"
86  module purge
87  module load devel/miniforge
88  module load devel/cuda/12.8
89
90  eval "$(command conda 'shell.bash' 'hook' 2> /dev/null)"
91  conda activate "$CONDA_ENV_NAME"
92
93  export PYTHONNOUSERSITE=1
94  export CUDA_DEVICE_MAX_CONNECTIONS=1
95  export MASTER_ADDR=$(scontrol show hostnames "$SLURM_JOB_NODELIST" | head
        -n 1)
96  export MASTER_PORT=29500
97
98  TARGET_NCCL="$CONDA_PREFIX/lib/libnccl.so.2"
99  if [ -f "$TARGET_NCCL" ]; then
100     export LD_PRELOAD="$TARGET_NCCL"
101     echo " [OVERRIDE] NCCL: Preloaded $TARGET_NCCL"
102 else
103     echo " [WARNING] Custom NCCL not found at $TARGET_NCCL. Using System/
            PyTorch default."
104 fi
105
106 #====================================================
107 # 3. PARAMETER CALCULATION
108 #====================================================
109
110 readonly GPUS_PER_NODE=${SLURM_GPUS_ON_NODE:-$SLURM_GPUS_PER_NODE}
111 readonly WORLD_SIZE=$((SLURM_NNODES * GPUS_PER_NODE))
112
113 if [[ $((WORLD_SIZE % (TP * PP * EP * CP))) -ne 0 ]]; then
114     echo "ERROR: Total GPUs ($WORLD_SIZE) not divisible by TP($TP) * PP(
            $PP) * EP($EP) * CP($CP)." >&2
115     exit 1
116 fi
117 DP=$((WORLD_SIZE / (TP * PP * EP * CP)))
```

```
118
119 SP_TAG=$(
120     if [[ "$SP" == "auto" && $TP -gt 1 ]]; then echo "SP1"
121     elif [[ "$SP" == "1" && $TP -gt 1 ]]; then echo "SP1"
122     else echo "SP0"
123     fi
124 )
125 RUN_DESC="${MODEL_SIZE}_${HW_NAME}_TP${TP}_PP${PP}_EP${EP}_CP${CP}_${
        SP_TAG}_DP${DP}_${SLURM_JOB_ID}"
126 readonly NCCL_LOG_DIR="/home/hd/hd_hd/hd_gn324/thesis/megatron_logs/
        nccl_logs/${RUN_DESC}"
127 readonly CHECKPOINT_PATH="$MEGATRON_ROOT_PATH/training/runs/checkpoints/
        $SLURM_JOB_ID"
128 readonly TENSORBOARD_LOGS_PATH="$CHECKPOINT_PATH/tensorboard"
129 readonly NSYS_OUTPUT_DIR="$NCCL_LOG_DIR"
130 mkdir -p "$CHECKPOINT_PATH" "$TENSORBOARD_LOGS_PATH" "slurm" "
        $NSYS_OUTPUT_DIR"
131
132 echo "--- Configuration ---"
133 echo " ID:            $RUN_DESC"
134 echo " Model:         $MODEL_SIZE"
135 echo " Conda Env:     $CONDA_ENV_NAME"
136 echo " Parallelism:   TP=$TP | PP=$PP | EP=$EP | CP=$CP | DP=$DP"
137 echo " Resources:     Nodes=$SLURM_NNODES | GPUs/Node=$GPUS_PER_NODE |
        World Size=$WORLD_SIZE"
138 echo " Master Addr:   $MASTER_ADDR:$MASTER_PORT"
139
140 IS_MOE=0
141 NUM_EXPERTS=0
142 EXPERT_INTERVAL=1
143
144 case "$MODEL_SIZE" in
145     "dense")
146         NUM_LAYERS=16; HIDDEN_SIZE=512; NUM_ATTENTION_HEADS=8; SEQ_LENGTH
                =1024
147         ;;
148     "dense_moe")
149         NUM_LAYERS=16; HIDDEN_SIZE=512; NUM_ATTENTION_HEADS=8; SEQ_LENGTH
                =1024
150         IS_MOE=1; NUM_EXPERTS=4; EXPERT_INTERVAL=2
151         ;;
152     *)
153         echo "ERROR: Unknown model size '$MODEL_SIZE'. Valid options: '
                dense', 'dense_moe'" >&2
154         exit 1
155         ;;
156 esac
157
158 GPT_MODEL_ARGS=(
159     --num-layers $NUM_LAYERS
160     --hidden-size $HIDDEN_SIZE
161     --num-attention-heads $NUM_ATTENTION_HEADS
162     --seq-length $SEQ_LENGTH
163     --max-position-embeddings $SEQ_LENGTH
164 )
165
166 if [[ $IS_MOE -eq 1 ]]; then
```

```
167     if [[ $EP -gt 1 ]] && [[ $((NUM_EXPERTS % EP)) -ne 0 ]]; then
168         echo "ERROR: Expert Parallelism (EP=$EP) must evenly divide
                number of experts ($NUM_EXPERTS)." >&2
169         echo "Valid EP values for $NUM_EXPERTS experts: 1, 2, 4$([
                $NUM_EXPERTS -eq 8 ] && echo ’, 8’)" >&2
170         exit 1
171     fi
172
173     GPT_MODEL_ARGS+=(
174         --num-experts $NUM_EXPERTS
175         --expert-model-parallel-size $EP
176         --moe-router-topk 2
177         --moe-router-load-balancing-type aux_loss
178         --moe-aux-loss-coeff 0.01
179         --moe-grouped-gemm
180         --disable-bias-linear
181     )
182
183     if [[ $EXPERT_INTERVAL -gt 1 ]]; then
184         GPT_MODEL_ARGS+=(--moe-layer-freq $EXPERT_INTERVAL)
185     fi
186
187     echo " [INFO] MoE enabled: $NUM_EXPERTS experts, EP=$EP, layer_freq=
            $EXPERT_INTERVAL"
188     echo " [INFO] MoE settings: bf16, bias disabled, grouped GEMM"
189 fi
190
191 case "$MODEL_SIZE" in
192     "dense")
193         MICRO_BATCH_SIZE=4
194         ;;
195     "dense_moe")
196         MICRO_BATCH_SIZE=4
197         ;;
198     *)
199         MICRO_BATCH_SIZE=1
200         ;;
201 esac
202 MICRO_BATCH_SIZE=$((MICRO_BATCH_SIZE > 0 ? MICRO_BATCH_SIZE : 1))
203
204 if [[ $((GLOBAL_BATCH_SIZE % (MICRO_BATCH_SIZE * DP))) -ne 0 ]]; then
205     echo "ERROR: Global batch size ($GLOBAL_BATCH_SIZE) not divisible by
            micro-batch ($MICRO_BATCH_SIZE) * DP ($DP)" >&2
206     exit 1
207 fi
208
209 GRADIENT_ACCUMULATION_STEPS=$((GLOBAL_BATCH_SIZE / (MICRO_BATCH_SIZE * DP
        )))
210
211 echo " Batch Config: Global=$GLOBAL_BATCH_SIZE | Micro=$MICRO_BATCH_SIZE
        | Grad Accum=$GRADIENT_ACCUMULATION_STEPS"
212
213 if [[ $PROFILE -eq 1 ]]; then
214     mkdir -p "${NCCL_LOG_DIR}"
215     export NCCL_DEBUG=INFO
216     export NCCL_DEBUG_SUBSYS=ALL
217     export NCCL_DEBUG_TIMESTAMP_FORMAT="%s.%6f"
```

```
218     export NCCL_DEBUG_TIMESTAMP_LEVELS=ALL
219     export NCCL_DEBUG_FILE="${NCCL_LOG_DIR}/nccl_debug_%h_%p.log"
220     export NCCL_TOPO_DUMP_FILE="${NCCL_LOG_DIR}/topo.xml"
221     export CUDNN_BENCHMARK=0
222     export NCCL_LAUNCH_ORDER_IMPLICIT=1
223 fi
224 
225 GPU_TOPO_FILE="${NCCL_LOG_DIR}/gpu_topology.txt"
226 echo "--- Saving GPU Topology to ${GPU_TOPO_FILE} ---"
227 {
228     echo "===================================================="
229     echo "GPU Topology Information"
230     echo "Job ID: $SLURM_JOB_ID"
231     echo "Date: $(date)"
232     echo "===================================================="
233     echo ""
234     echo "--- GPU Info (nvidia-smi) ---"
235     nvidia-smi --query-gpu=timestamp,name,pci.bus_id,driver_version,
            memory.total,memory.free --format=csv
236     echo ""
237     echo "--- NVLink Topology ---"
238     nvidia-smi topo -m
239     echo ""
240     echo "===================================================="
241 } > "$GPU_TOPO_FILE"
242 
243 CONFIG_METADATA_FILE="${NCCL_LOG_DIR}/run_config.txt"
244 cat > "$CONFIG_METADATA_FILE" << EOF
245 =================================================
246 CONFIGURATION METADATA
247 =================================================
248 Job ID:         $SLURM_JOB_ID
249 Date:           $(date)
250 Framework:      Megatron-LM (Torchrun)
251 GPU Type:       $HW_NAME
252 Model Size:     $MODEL_SIZE
253 Model Type:     $([ $IS_MOE -eq 1 ] && echo "MoE (Experts=$NUM_EXPERTS,
        layer_freq=$EXPERT_INTERVAL)" || echo "Dense")
254 Parallelism:    TP=$TP, PP=$PP, EP=$EP, CP=$CP, DP=$DP
255 Seq Parallel:   $([ "$SP" == "auto" ] && echo "AUTO" || echo "$SP")
256 Batch Sizes:    Global=$GLOBAL_BATCH_SIZE, Micro=$MICRO_BATCH_SIZE
257 Grad Accum:     $GRADIENT_ACCUMULATION_STEPS
258 Total GPUs:     $WORLD_SIZE (Nodes=$SLURM_NNODES x GPUs=$GPUS_PER_NODE)
259 Iterations:     $ITERATIONS
260 Model Config:   Layers=$NUM_LAYERS, Hidden=$HIDDEN_SIZE, Heads=
        $NUM_ATTENTION_HEADS, SeqLen=$SEQ_LENGTH
261 Precision:      $([ $IS_MOE -eq 1 ] && echo "bf16 (MoE required)" || echo
         "fp16")
262 Optimization:   ${OPT_FLAGS:-"None"}
263 =================================================
264 EOF
265 
266 #===================================================
267 # 4. ARGUMENT ASSEMBLY
268 #===================================================
269 
270 if [[ $IS_MOE -eq 1 ]]; then
```

```
271     PRECISION_FLAG="--bf16"
272     echo " [INFO] Using bf16 precision (required for MoE grouped GEMM)"
273 else
274     PRECISION_FLAG="--fp16"
275     echo " [INFO] Using fp16 precision"
276 fi
277 
278 TRAINING_ARGS=(
279     --micro-batch-size "$MICRO_BATCH_SIZE"
280     --global-batch-size "$GLOBAL_BATCH_SIZE"
281     --train-iters "$ITERATIONS"
282     --weight-decay 0.1
283     --adam-beta1 0.9
284     --adam-beta2 0.95
285     --init-method-std 0.006
286     --clip-grad 1.0
287     $PRECISION_FLAG
288     --lr 6.0e-5
289     --lr-decay-style cosine
290     --min-lr 6.0e-6
291     --lr-warmup-fraction .001
292     --lr-decay-iters 430000
293     --seed "$REPRODUCIBILITY_SEED"
294 )
295 
296 MODEL_PARALLEL_ARGS=(
297     --tensor-model-parallel-size "$TP"
298     --pipeline-model-parallel-size "$PP"
299 )
300 
301 if [[ "$SP" == "auto" ]]; then
302     if [[ $TP -gt 1 ]]; then
303         MODEL_PARALLEL_ARGS+=(--sequence-parallel)
304         echo " [INFO] Sequence parallelism AUTO-ENABLED (TP=$TP)"
305     else
306         echo " [INFO] Sequence parallelism AUTO-DISABLED (TP=1)"
307     fi
308 elif [[ "$SP" == "1" ]]; then
309     if [[ $TP -gt 1 ]]; then
310         MODEL_PARALLEL_ARGS+=(--sequence-parallel)
311         echo " [INFO] Sequence parallelism FORCE-ENABLED"
312     else
313         echo " [WARNING] Cannot enable SP with TP=1. Ignoring SP=1."
314     fi
315 elif [[ "$SP" == "0" ]]; then
316     echo " [INFO] Sequence parallelism FORCE-DISABLED"
317 else
318     echo " [ERROR] Invalid SP argument: $SP. Use 'auto', '1', or '0'."
         >&2
319     exit 1
320 fi
321 
322 DATA_ARGS=(
323     --data-path "${DATA_PATH_PREFIX}/fineweb_text_document"
324     --vocab-file "${DATA_PATH_PREFIX}/vocab.json"
325     --merge-file "${DATA_PATH_PREFIX}/merges.txt"
326     --split 949,50,1
```

```
327 )
328
329 EVAL_ARGS=(
330     --log-interval 1
331     --eval-interval 1000
332     --eval-iters 0
333     --tensorboard-dir "$TENSORBOARD_LOGS_PATH"
334     --log-timers-to-tensorboard
335     --tensorboard-log-interval 1
336 )
337
338 CHECKPOINT_ARGS=()
339 if [[ "$MODEL_SIZE" == "dense_moe" ]]; then
340     CHECKPOINT_ARGS=(--recompute-activations)
341     echo " [INFO] Activation checkpointing enabled for $MODEL_SIZE"
342 fi
343
344 ALL_MEGATRON_ARGS=(
345     "${GPT_MODEL_ARGS[@]}"
346     "${TRAINING_ARGS[@]}"
347     "${MODEL_PARALLEL_ARGS[@]}"
348     "${CHECKPOINT_ARGS[@]}"
349     "${DATA_ARGS[@]}"
350     "${EVAL_ARGS[@]}"
351 )
352
353 if [[ -n "$OPT_FLAGS" ]]; then
354     read -ra OPT_FLAGS_ARRAY <<< "$OPT_FLAGS"
355     ALL_MEGATRON_ARGS+=("${OPT_FLAGS_ARRAY[@]}")
356     echo " [INFO] Optimization flags enabled: $OPT_FLAGS"
357 fi
358
359 TRAINING_SCRIPT_PATH="$MEGATRON_ROOT_PATH/megatron/pretrain_gpt.py"
360
361 COMMAND_TO_RUN=(
362     torchrun
363     --nproc_per_node="$GPUS_PER_NODE"
364     --nnodes="$SLURM_NNODES"
365     --rdzv_id="$SLURM_JOB_ID"
366     --rdzv_backend=c10d
367     --rdzv_endpoint="$MASTER_ADDR:$MASTER_PORT"
368     "$TRAINING_SCRIPT_PATH"
369     "${ALL_MEGATRON_ARGS[@]}"
370 )
371
372 #=====================================================
373 # 5. EXECUTION
374 #=====================================================
375
376 echo "--- Starting Training ---"
377
378 set +e
379
380 if [[ $PROFILE -eq 1 ]]; then
381     echo "Mode: NSIGHT PROFILING"
382
383     NSYS_CMD=(
```

```
384         nsys profile
385         -o "${NSYS_OUTPUT_DIR}/profile_rank_%h_%p"
386         --trace=cuda,nvtx
387         --sample=none
388         --cpuctxsw=none
389         --force-overwrite=true
390         --capture-range=none
391         --stop-on-exit=true
392         --delay=0
393         --duration=0
394         --stats=false
395         --export=none
396     )
397 
398     if [[ $SLURM_NNODES -eq 1 ]]; then
399         "${NSYS_CMD[@]}" "${COMMAND_TO_RUN[@]}"
400     else
401         srun "${NSYS_CMD[@]}" "${COMMAND_TO_RUN[@]}"
402     fi
403 else
404     echo "Mode: STANDARD EXECUTION"
405     srun "${COMMAND_TO_RUN[@]}"
406 fi
407 
408 set -e
409 
410 #====================================================
411 # 6. POST-PROCESSING (Profiling Only)
412 #====================================================
413 
414 convert_to_sqlite() {
415     local nsys_rep_file="$1"
416     local sqlite_file="${nsys_rep_file%.nsys-rep}.sqlite"
417 
418     echo "Converting: ${nsys_rep_file} -> ${sqlite_file}"
419     if timeout 600 nsys export --type sqlite -o "${sqlite_file}" "${
         nsys_rep_file}"; then
420         echo "Success: ${sqlite_file}"
421     else
422         echo "ERROR: Conversion failed for ${nsys_rep_file}"
423     fi
424 }
425 
426 if [[ $PROFILE -eq 1 ]]; then
427     echo "--- Processing Profile Data ---"
428     shopt -s nullglob
429     nsys_files=( "${NSYS_OUTPUT_DIR}"/*.nsys-rep )
430     shopt -u nullglob
431 
432     if [ ${#nsys_files[@]} -eq 0 ]; then
433         echo "WARNING: No .nsys-rep files found in ${NSYS_OUTPUT_DIR}."
434     else
435         for nsys_file in "${nsys_files[@]}"; do
436             convert_to_sqlite "$nsys_file"
437         done
438     fi
439 fi
```

```
440
441 echo "=================================================="
442 echo "Job Complete."
443 echo "=================================================="
```

Listing a.2: Training Script

BIBLIOGRAPHY

[1] Inc Advanced Micro Devices. *AMD CDNA 3 Architecture*. 2023. URL: https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/white-papers/amd-cdna-3-white-paper.pdf.

[2] Inc Advanced Micro Devices. *AMD CDNA 4 Architecture*. 2025. URL: https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/white-papers/amd-cdna-4-architecture-whitepaper.pdf.

[3] Tom B. Brown et al. *Language Models Are Few-Shot Learners*. July 2020. DOI: 10.48550/arXiv.2005.14165. arXiv: 2005.14165 [cs]. URL: http://arxiv.org/abs/2005.14165 (visited on 01/15/2026).

[4] Ziteng Chen et al. *An Efficient, Reliable and Observable Collective Communication Library in Large-scale GPU Training Clusters*. Oct. 2025. DOI: 10.48550/arXiv.2510.00991. arXiv: 2510.00991 [cs]. URL: http://arxiv.org/abs/2510.00991 (visited on 01/13/2026).

[5] Aakanksha Chowdhery et al. *PaLM: Scaling Language Modeling with Pathways*. Oct. 2022. DOI: 10.48550/arXiv.2204.02311. arXiv: 2204.02311 [cs]. URL: http://arxiv.org/abs/2204.02311 (visited on 01/15/2026).

[6] Gheorghe Comanici et al. *Gemini 2.5: Pushing the Frontier with Advanced Reasoning, Multimodality, Long Context, and Next Generation Agentic Capabilities*. Dec. 2025. DOI: 10.48550/arXiv.2507.06261. arXiv: 2507.06261 [cs]. URL: http://arxiv.org/abs/2507.06261 (visited on 01/10/2026).

[7] Intel Corporation. *Habani Gaudi 3 Technical Paper*. 2025. URL: https://cdrdv2-public.intel.com/817486/gaudi-3-ai-accelerator-white-paper.pdf.

[8] NVIDIA Corporation. *NVIDIA A100 Tensor Core GPU Architecture*. 2020. URL: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[9] NVIDIA Corporation. *NVIDIA H100 Tensor Core GPU Architecture*. 2022. URL: https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c.

[10] NVIDIA Corporation. *ncclKernel Bytes Unavailable in Nsight Systems Stats View*. 2023. URL: https://forums.developer.nvidia.com/t/ncclkernel-bytes-unavailable-in-nsight-systems-stats-view/273981.

[11] NVIDIA Corporation. *NVIDIA Blackwell Technical Brief*. 2024. URL: https://resources.nvidia.com/en-us-blackwell-architecture.

[12] NVIDIA Corporation. *Context Parallelism*. URL: https://docs.nvidia.com/megatron-core/developer-guide/latest/api-guide/context_parallel.html.

[13] NVIDIA Corporation. *Introduction to the NVIDIA DGX A100 System*. URL: https://docs.nvidia.com/dgx/dgxa100-user-guide/introduction-to-dgxa100.html.

[14] NVIDIA Corporation. *Mixture of Experts*. URL: https://docs.nvidia.com/megatron-core/developer-guide/latest/api-guide/moe.html.

[15] NVIDIA Corporation. *NCCL*. URL: https://github.com/NVIDIA/nccl.

[16] NVIDIA Corporation. *NCCL Documentation*. URL: https://docs.nvidia.com/deeplearning/nccl/.

[17] NVIDIA Corporation. *NCCL Tests*. URL: https://github.com/nvidia/nccl-tests.

[18] NVIDIA Corporation. *NDR Overview*. URL: https://docs.nvidia.com/dgx-superpod/design-guide-cabling-data-centers/latest/ndr-overview.html.

[19] NVIDIA Corporation. *Nsight Systems Documentation*. URL: https://docs.nvidia.com/nsight-systems/.

[20] NVIDIA Corporation. *NVIDIA DGX B300*. URL: https://www.nvidia.com/en-us/data-center/dgx-b300/.

[21] NVIDIA Corporation. *NVIDIA DGX H100*. URL: NVIDIA%20DGX%20H100.

[22] NVIDIA Corporation. *NVIDIA Rubin Press Release*. URL: https://nvidianews.nvidia.com/news/rubin-platform-ai-supercomputer.

[23] NVIDIA Corporation. *Parallelisms*. URL: https://docs.nvidia.com/megatron-core/developer-guide/latest/user-guide/parallelism-guide.html.

[24] NVIDIA Corporation and Sylvain Jeaugey. *Scaling Deep Learning Training with NCCL*. 2018. URL: https://developer.nvidia.com/blog/scaling-deep-learning-training-nccl/.

[25] "Atlas of Protein Sequence and Structure. Vol. 5, Suppl. 3." In: ed. by Margaret O. Dayhoff. Washington, D.C: National Biomedical Research Foundation, 1979. ISBN: 978-0-912466-07-1.

[26] DeepSeek-AI et al. *DeepSeek LLM: Scaling Open-Source Language Models with Longtermism*. Jan. 5, 2024. DOI: 10.48550/arXiv.2401.02954. arXiv: 2401.02954 [cs]. URL: http://arxiv.org/abs/2401.02954 (visited on 09/17/2025). Pre-published.

[27] DeepSeek-AI et al. *DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model*. June 19, 2024. DOI: 10.48550/arXiv.2405.04434. arXiv: 2405.04434 [cs]. URL: http://arxiv.org/abs/2405.04434 (visited on 09/17/2025). Pre-published.

[28] DeepSeek-AI et al. *DeepSeek-V3 Technical Report*. Version 1. Dec. 27, 2024. DOI: 10.48550/arXiv.2412.19437. arXiv: 2412.19437 [cs]. URL: http://arxiv.org/abs/2412.19437 (visited on 05/24/2025). Pre-published.

[29] William Fedus, Barret Zoph, and Noam Shazeer. *Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity*. June 2022. DOI: 10.48550/arXiv.2101.03961. arXiv: 2101.03961 [cs]. URL: http://arxiv.org/abs/2101.03961 (visited on 01/13/2026).

[30] Denis Foley and John Danskin. "Ultra-Performance Pascal GPU and NVLink Interconnect." In: *IEEE Micro* 37.2 (Mar. 2017), pp. 7–17. ISSN: 0272-1732, 1937-4143. DOI: 10.1109/MM.2017.37. URL: https://ieeexplore.ieee.org/document/7924274/ (visited on 01/13/2026).

[31] Seokjin Go, Joongun Park, Spandan More, Hanjiang Wu, Irene Wang, Aaron Jezghani, Tushar Krishna, and Divya Mahajan. "Characterizing the Efficiency of Distributed Training: A Power, Performance, and Thermal Perspective." In: *Proceedings of the 2025 58th IEEE/ACM International Symposium on Microarchitecture*. Seoul Korea: ACM, Oct. 2025, pp. 626–642. ISBN: 979-8-4007-1573-0. DOI: 10.1145/3725843.3756111. URL: https://dl.acm.org/doi/10.1145/3725843.3756111 (visited on 11/21/2025).

[32] Osamu Gotoh. "An Improved Algorithm for Matching Biological Sequences." In: *Journal of Molecular Biology* 162.3 (Dec. 1982), pp. 705–708. ISSN: 00222836. DOI: 10.1016/0022-2836(82)90398-9. URL: https://linkinghub.elsevier.com/retrieve/pii/0022283682903989 (visited on 01/13/2026).

[33] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. *Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour*. Apr. 2018. DOI: 10.48550/arXiv.1706.02677. arXiv: 1706.02677 [cs]. URL: http://arxiv.org/abs/1706.02677 (visited on 01/13/2026).

[34] Bagus Hanindhito, Bhavesh Patel, and Lizy K. John. “Bandwidth Characterization of DeepSpeed on Distributed Large Language Model Training.” In: *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Indianapolis, IN, USA: IEEE, May 2024, pp. 241–256. ISBN: 979-8-3503-7638-8. DOI: 10.1109/ISPASS61541.2024.00031. URL: https://ieeexplore.ieee.org/document/10590030/ (visited on 09/22/2025).

[35] Sam Hassan. *Understanding LLM GPUs Clusters Fabrics Traffic For Networkers*. 2025. URL: https://www.dell.com/en-us/blog/understanding-llm-gpus-clusters-fabrics-traffic-for-networkers/.

[36] S Henikoff and J G Henikoff. “Amino Acid Substitution Matrices from Protein Blocks.” In: *Proceedings of the National Academy of Sciences* 89.22 (Nov. 1992), pp. 10915–10919. ISSN: 0027-8424, 1091-6490. DOI: 10.1073/pnas.89.22.10915. URL: https://pnas.org/doi/full/10.1073/pnas.89.22.10915 (visited on 01/06/2026).

[37] Jordan Hoffmann et al. *Training Compute-Optimal Large Language Models*. Mar. 2022. DOI: 10.48550/arXiv.2203.15556. arXiv: 2203.15556 [cs]. URL: http://arxiv.org/abs/2203.15556 (visited on 01/15/2026).

[38] Zhiyi Hu, Siyuan Shen, Tommaso Bonato, Sylvain Jeaugey, Cedell Alexander, Eric Spada, James Dinan, Jeff Hammond, and Torsten Hoefler. *Demystifying NCCL: An In-depth Analysis of GPU Communication Protocols and Algorithms*. July 2025. DOI: 10.48550/arXiv.2507.04786. arXiv: 2507.04786 [cs]. URL: http://arxiv.org/abs/2507.04786 (visited on 09/22/2025).

[39] Yanping Huang et al. *GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism*. July 2019. DOI: 10.48550/arXiv.1811.06965. arXiv: 1811.06965 [cs]. URL: http://arxiv.org/abs/1811.06965 (visited on 01/13/2026).

[40] Mikhail Isaev, Nic Mcdonald, Larry Dennison, and Richard Vuduc. “Calculon: A Methodology and Tool for High-Level Co-Design of Systems and Large Language Models.” In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Denver CO USA: ACM, Nov. 2023, pp. 1–14. ISBN: 979-8-4007-0109-2. DOI: 10.1145/3581784.3607102. URL: https://dl.acm.org/doi/10.1145/3581784.3607102 (visited on 01/15/2026).

[41] Mohammad Kefah Taha Issa, Muhammad Aditya Sasongko, Ilyas Turimbetov, Javid Baydamirli, Doğan Sağbili, and Didem Unat. “Snoopie: A Multi-GPU Communication Profiler and Visualizer.” In: *Proceedings of the 38th ACM International Conference on Supercomputing*. Kyoto Japan: ACM, May 2024, pp. 525–536. ISBN: 979-8-4007-0610-3. DOI: 10.1145/3650200.3656597. URL:

https://dl.acm.org/doi/10.1145/3650200.3656597 (visited on 02/05/2026).

[42] Ziheng Jiang et al. *MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs*. 2024. DOI: 10.48550/ARXIV.2402.15627. URL: https://arxiv.org/abs/2402.15627 (visited on 11/21/2025).

[43] Norman P. Jouppi et al. *TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings*. Apr. 2023. DOI: 10.48550/arXiv.2304.01433. arXiv: 2304.01433 [cs]. URL: http://arxiv.org/abs/2304.01433 (visited on 01/15/2026).

[44] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. *Scaling Laws for Neural Language Models*. Jan. 2020. DOI: 10.48550/arXiv.2001.08361. arXiv: 2001.08361 [cs]. URL: http://arxiv.org/abs/2001.08361 (visited on 01/15/2026).

[45] Joyjit Kundu, Wenzhe Guo, Ali BanaGozar, Udari De Alwis, Sourav Sengupta, Puneet Gupta, and Arindam Mallik. "Performance Modeling and Workload Analysis of Distributed Large Language Model Training and Inference." In: *2024 IEEE International Symposium on Workload Characterization (IISWC)*. Vancouver, BC, Canada: IEEE, Sept. 2024, pp. 57–67. ISBN: 979-8-3503-5603-8. DOI: 10.1109/IISWC63097.2024.00015. URL: https://ieeexplore.ieee.org/document/10763669/ (visited on 11/21/2025).

[46] Sunwoo Lee, Dipendra Jha, Ankit Agrawal, Alok Choudhary, and Wei-keng Liao. "Parallel Deep Convolutional Neural Network Training by Exploiting the Overlapping of Computation and Communication." In: *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. Jaipur: IEEE, Dec. 2017, pp. 183–192. ISBN: 978-1-5386-2293-3. DOI: 10.1109/HiPC.2017.00030. URL: http://ieeexplore.ieee.org/document/8287749/ (visited on 01/15/2026).

[47] Qingyuan Li, Bo Zhang, Liang Ye, Yifan Zhang, Wei Wu, Yerui Sun, Lin Ma, and Yuchen Xie. *Flash Communication: Reducing Tensor Parallelization Bottleneck for Fast Large Language Model Inference*. Dec. 2024. DOI: 10.48550/arXiv.2412.04964. arXiv: 2412.04964 [cs]. URL: http://arxiv.org/abs/2412.04964 (visited on 01/13/2026).

[48] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. "Sequence Parallelism: Long Sequence Training from System Perspective." In: *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Toronto, Canada: Association for Computational Linguistics, 2023, pp. 2391–2404. DOI: 10.18653/v1/2023.acl-l

ong.134. URL: https://aclanthology.org/2023.acl-long.134 (visited on 01/15/2026).

[49] Mingyu Liang, Hiwot Tadese Kassa, Wenyin Fu, Brian Coutinho, Louis Feng, and Christina Delimitrou. *Lumos: Efficient Performance Modeling and Estimation for Large-scale LLM Training*. Apr. 2025. DOI: 10.48550/arXiv.2504.09307. arXiv: 2504.09307 [cs]. URL: http://arxiv.org/abs/2504.09307 (visited on 01/21/2026).

[50] Dennis Liu et al. *MoE Parallel Folding: Heterogeneous Parallelism Mappings for Efficient Large-Scale MoE Model Training with Megatron Core*. Apr. 2025. DOI: 10.48550/arXiv.2504.14960. arXiv: 2504.14960 [cs]. URL: http://arxiv.org/abs/2504.14960 (visited on 02/04/2026).

[51] Google LLC. *Tensorflow Profiler*. URL: https://www.tensorflow.org/guide/profiler.

[52] Google LLC, Amin Vahdat, and Mark Lohmeyer. *Enabling Next-Generation AI Workloads: Announcing TPU V5p and AI Hypercomputer*. URL: https://cloud.google.com/blog/products/ai-machine-learning/introducing-cloud-tpu-v5p-and-ai-hypercomputer.

[53] Paulius Micikevicius et al. *Mixed Precision Training*. Feb. 2018. DOI: 10.48550/arXiv.1710.03740. arXiv: 1710.03740 [cs]. URL: http://arxiv.org/abs/1710.03740 (visited on 01/15/2026).

[54] Deepak Narayanan et al. "Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. St. Louis Missouri: ACM, Nov. 2021, pp. 1–15. ISBN: 978-1-4503-8442-1. DOI: 10.1145/3458817.3476209. URL: https://dl.acm.org/doi/10.1145/3458817.3476209 (visited on 09/22/2025).

[55] Saul B. Needleman and Christian D. Wunsch. "A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins." In: *Journal of Molecular Biology* 48.3 (Mar. 1970), pp. 443–453. ISSN: 00222836. DOI: 10.1016/0022-2836(70)90057-4. URL: https://linkinghub.elsevier.com/retrieve/pii/0022283670900574 (visited on 01/06/2026).

[56] OpenAI. *GPT-5*. URL: https://openai.com/index/introducing-gpt-5/.

[57] OpenAI et al. *GPT-4 Technical Report*. Mar. 2024. DOI: 10.48550/arXiv.2303.08774. arXiv: 2303.08774 [cs]. URL: http://arxiv.org/abs/2303.08774 (visited on 01/22/2026).

[58] Shuo Ouyang, Dezun Dong, Yemao Xu, and Liquan Xiao. "Communication Optimization Strategies for Distributed Deep Neural Network Training: A Survey." In: (2020). DOI: 10.48550/ARXIV.2003.03009. URL: https://arxiv.org/abs/2003.03009 (visited on 01/07/2026).

[59] Pitch Patarasuk and Xin Yuan. "Bandwidth Optimal All-Reduce Algorithms for Clusters of Workstations." In: *Journal of Parallel and Distributed Computing* 69.2 (Feb. 2009), pp. 117–124. ISSN: 07437315. DOI: 10.1016/j.jpdc.2008.09.002. URL: https://linkinghub.elsevier.com/retrieve/pii/S0743731508001767 (visited on 01/13/2026).

[60] Guilherme Penedo, Hynek Kydlíček, Loubna Ben allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, and Thomas Wolf. *The FineWeb Datasets: Decanting the Web for the Finest Text Data at Scale*. Oct. 2024. DOI: 10.48550/arXiv.2406.17557. arXiv: 2406.17557 [cs]. URL: http://arxiv.org/abs/2406.17557 (visited on 01/06/2026).

[61] Rolf Rabenseifner. "Optimization of Collective Reduction Operations." In: *Computational Science - ICCS 2004*. Ed. by Takeo Kanade et al. Vol. 3036. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 1–9. ISBN: 978-3-540-22114-2 978-3-540-24685-5. DOI: 10.1007/978-3-540-24685-5_1. URL: http://link.springer.com/10.1007/978-3-540-24685-5_1 (visited on 01/13/2026).

[62] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. "Improving Language Understanding by Generative Pre-Training." In: (). URL: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf.

[63] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. "Language Models Are Unsupervised Multitask Learners." In: (). URL: https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

[64] Shivam Raikundalia. *PyTorch Profiler*. 2025. URL: https://docs.pytorch.org/tutorials/recipes/recipes/profiler_recipe.html.

[65] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models." In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. Atlanta, GA, USA: IEEE, Nov. 2020, pp. 1–16. ISBN: 978-1-7281-9998-6. DOI: 10.1109/SC41405.2020.00024. URL: https://ieeexplore.ieee.org/document/9355301/ (visited on 11/22/2025).

[66] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. "DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters." In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. Virtual Event CA USA: ACM, Aug. 2020, pp. 3505–3506. ISBN: 978-1-4503-7998-4. DOI: 10.1145/3394486.3406703. URL: https://dl.acm.org/doi/10.1145/3394486.3406703 (visited on 11/22/2025).

[67] Ties Robroek, Ehsan Yousefzadeh-Asl-Miandoab, and Pınar Tözün. *An Analysis of Collocation on GPUs for Deep Learning Training*. Apr. 2023. DOI: 10.48550/arXiv.2209.06018. arXiv: 2209.06018 [cs]. URL: http://arxiv.org/abs/2209.06018 (visited on 01/15/2026).

[68] Jaime Sevilla, Lennart Heim, Anson Ho, Tamay Besiroglu, Marius Hobbhahn, and Pablo Villalobos. "Compute Trends Across Three Eras of Machine Learning." In: (2022). DOI: 10.48550/ARXIV.2202.05924. URL: https://arxiv.org/abs/2202.05924 (visited on 01/07/2026).

[69] Gilad Shainer, Pak Lui, and Tong Liu. "The Development of Mellanox/NVIDIA GPUDirect over InfiniBand: A New Model for GPU to GPU Communications." In: *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*. Salt Lake City Utah: ACM, July 2011, pp. 1–1. ISBN: 978-1-4503-0888-5. DOI: 10.1145/2016741.2016769. URL: https://dl.acm.org/doi/10.1145/2016741.2016769 (visited on 01/07/2026).

[70] 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*. Jan. 2017. DOI: 10.48550/arXiv.1701.06538. arXiv: 1701.06538 [cs]. URL: http://arxiv.org/abs/1701.06538 (visited on 01/13/2026).

[71] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. *Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism*. 2019. DOI: 10.48550/ARXIV.1909.08053. URL: https://arxiv.org/abs/1909.08053 (visited on 11/21/2025).

[72] Min Si et al. *Collective Communication for 100k+ GPUs*. Jan. 2026. DOI: 10.48550/arXiv.2510.20171. arXiv: 2510.20171 [cs]. URL: http://arxiv.org/abs/2510.20171 (visited on 01/13/2026).

[73] Prajwal Singhania, Siddharth Singh, Lannie Dalton Hough, Akarsh Srivastava, Harshitha Menon, Charles Fredrick Jekel, and Abhinav Bhatele. *LLM Inference Beyond a Single Node: From Bottlenecks to Mitigations with Fast All-Reduce Communication*. Dec. 2025. DOI: 10.48550/arXiv.2511.09557. arXiv: 2511.09557

[cs]. URL: http://arxiv.org/abs/2511.09557 (visited on 01/21/2026).

[74] T.F. Smith and M.S. Waterman. "Identification of Common Molecular Subsequences." In: *Journal of Molecular Biology* 147.1 (Mar. 1981), pp. 195–197. ISSN: 00222836. DOI: 10.1016/0022-2836(81)90087-5. URL: https://linkinghub.elsevier.com/retrieve/pii/0022283681900875 (visited on 01/06/2026).

[75] Gemini Team et al. *Gemini: A Family of Highly Capable Multimodal Models*. May 2025. DOI: 10.48550/arXiv.2312.11805. arXiv: 2312.11805 [cs]. URL: http://arxiv.org/abs/2312.11805 (visited on 01/10/2026).

[76] Mellanox Technologies. *Introducing 200G HDR InfiniBand Solutions*. 2019. URL: https://network.nvidia.com/files/doc-2020/wp-introducing-200g-hdr-infiniband-solutions.pdf.

[77] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. "Optimization of Collective Communication Operations in MPICH." In: *The International Journal of High Performance Computing Applications* 19.1 (Feb. 2005), pp. 49–66. ISSN: 1094-3420, 1741-2846. DOI: 10.1177/1094342005051521. URL: https://journals.sagepub.com/doi/10.1177/1094342005051521 (visited on 01/13/2026).

[78] Hugo Touvron et al. *Llama 2: Open Foundation and Fine-Tuned Chat Models*. July 2023. DOI: 10.48550/arXiv.2307.09288. arXiv: 2307.09288 [cs]. URL: http://arxiv.org/abs/2307.09288 (visited on 01/15/2026).

[79] Hugo Touvron et al. *LLaMA: Open and Efficient Foundation Language Models*. Feb. 2023. DOI: 10.48550/arXiv.2302.13971. arXiv: 2302.13971 [cs]. URL: http://arxiv.org/abs/2302.13971 (visited on 01/15/2026).

[80] Didem Unat, Ilyas Turimbetov, Mohammed Kefah Taha Issa, Doğan Sağbili, Flavio Vella, Daniele De Sensi, and Ismayil Ismayilov. *The Landscape of GPU-Centric Communication*. Sept. 2024. DOI: 10.48550/arXiv.2409.09874. arXiv: 2409.09874 [cs]. URL: http://arxiv.org/abs/2409.09874 (visited on 01/15/2026).

[81] Ioannis Vardas, Ruben Laso Rodriguez, and Majid Salimi Beni. "Ncclsee: A Lightweight Profiling Tool for NCCL." In: 2025. URL: https://repositum.tuwien.at/bitstream/20.500.12708/218557/1/Vardas-2025-ncclsee%20A%20Lightweight%20Profiling%20Tool%20for%20NCCL-vor.pdf.

[82] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need*. Aug. 2023. DOI: 10.48550/arXiv.1706.03762. arXiv: 1706.03762 [cs]. URL: http://arxiv.org/abs/1706.03762 (visited on 01/15/2026).

[83] Haiquan Wang, Chaoyi Ruan, Jia He, Jiaqi Ruan, Chengjie Tang, Xiaosong Ma, and Cheng Li. *Hiding Communication Cost in Distributed LLM Training via Micro-batch Co-execution*. Nov. 2024. DOI: 10.48550/arXiv.2411.15871. arXiv: 2411.15871 [cs]. URL: http://arxiv.org/abs/2411.15871 (visited on 01/13/2026).

[84] Jason Wei et al. *Emergent Abilities of Large Language Models*. Oct. 2022. DOI: 10.48550/arXiv.2206.07682. arXiv: 2206.07682 [cs]. URL: http://arxiv.org/abs/2206.07682 (visited on 01/22/2026).

[85] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. *Large Batch Optimization for Deep Learning: Training BERT in 76 Minutes*. Jan. 2020. DOI: 10.48550/arXiv.1904.00962. arXiv: 1904.00962 [cs]. URL: http://arxiv.org/abs/1904.00962 (visited on 01/15/2026).

[86] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing. *Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters*. June 2017. DOI: 10.48550/arXiv.1706.03292. arXiv: 1706.03292 [cs]. URL: http://arxiv.org/abs/1706.03292 (visited on 01/15/2026).

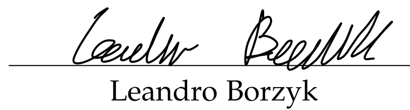
[87] Bohan Zhao, Guang Yang, Shuo Chen, Ruitao Liu, Tingrui Zhang, Yongchao He, and Wei Xu. *MegatronApp: Efficient and Comprehensive Management on Distributed LLM Training*. July 2025. DOI: 10.48550/arXiv.2507.19845. arXiv: 2507.19845 [cs]. URL: http://arxiv.org/abs/2507.19845 (visited on 01/13/2026).

[88] Yanli Zhao et al. *PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel*. Sept. 2023. DOI: 10.48550/arXiv.2304.11277. arXiv: 2304.11277 [cs]. URL: http://arxiv.org/abs/2304.11277 (visited on 01/13/2026).

ERKLÄRUNG

Ich versichere, dass ich diese Arbeit selbständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, den 20/02/2026



Leandro Borzyk