POLITECNICO DI TORINO

POLITECNICO DI TORINO

Master Degree course in Communications and Computer Networks Engineering

Master Degree Thesis

Distributed AI fabrics: a network-side perspective

Supervisors
Prof. Paolo Giaccone
Prof. Emilio Leonardi

Candidate
Francesco Camilli

Academic Year 2025-2026

Acknowledgements

Abstract

The rapid scaling of Large Language Models (LLMs) has transformed distributed training into a network-intensive workload, where communication efficiency increasingly dominates overall performance. As model size grows, the exchange of gradients between computing devices becomes a critical bottleneck, especially in geographically distributed or resource-constrained environments.

This thesis investigates the impact of network latency and bandwidth constraints on decentralized LLM training, with a specific focus on the Ring All-Reduce gradient synchronization algorithm. To analyze this phenomenon, a controlled experimental environment was implemented using Docker containerization on a single physical server, where multiple nodes are interconnected through a software-defined network to emulate a decentralized topology.

The study characterizes network traffic at packet level through tcpdump and Wireshark captures. A sensitivity analysis was conducted by varying key parameters including batch size, floating-point precision (bit-per-parameter), and world size, to investigate their influence on communication volume and synchronization time. By combining packet-level traffic measurements with theoretical performance analysis, we assess how computation and communication interact during each training step.

Results demonstrate the direct proportionality between parameter precision and communication payload, the trade-off between synchronization frequency and batch scaling, and the linear growth of communication overhead as world size increases. The findings validate the theoretical decomposition of training step time into compute and synchronization components, highlighting the conditions under which network constraints dominate training performance.

This work contributes a reproducible methodology for analyzing distributed LLM training traffic and provides quantitative insights into the relationship between gradient synchronization dynamics and network resource provisioning in decentralized AI infrastructures.

Contents

Chapter 1

Introduction

Artificial Intelligence (AI) is a computer science field that deals with creating systems capable of understanding human language, reasoning, and solving complex problems. Thanks to its ability to adapt to different scenarios, AI is employed in the most diverse areas today. To achieve this level of intelligence, these models must perform a learning phase called *training*, where they retrieve information from massive amounts of data. In recent years, as models have become incredibly large, the computational power required to train them has grown exponentially.

Today, training a state-of-the-art AI model on a single computer is impossible. The traditional solution has been to build massive datacenters housing thousands of specialized processors. However, this centralized approach is hitting a physical limit: energy consumption. The newest AI training facilities require an enormous amount of electricity, comparable to the energy consumption of a small city. Finding a single location capable of providing this much power is becoming practically impossible without overloading the local power grid.

To cope with this, the AI industry is forced to move towards a distributed environment. Distributing AI means splitting the enormous computational workload across multiple processors. This distribution can happen on two main levels: locally, connecting thousands of computers within a single datacenter, or geographically, connecting several smaller datacenters located hundreds or thousands of kilometers apart near available power plants. Major technology vendors are already heavily investing in this vision, offering commercial networking products designed to interconnect these distant facilities so they can cooperate as a single, giant AI factory.

While distributing the computation solves the energy problem, it introduces a new major challenge. Because of its distributed nature, the processors must constantly communicate over the network to share their partial findings and update the shared model. In geographically distributed scenarios, the network plays a crucial role: if the connection is slow or introduces too much delay, the processors will spend more time waiting for data to arrive than actually performing useful computations. The network itself becomes the bottleneck of the whole process.

In this work, we address the effect that network constraints have on decentralized AI

training and we propose an experimental framework to analyze it. Our system architecture uses a controlled, reproducible environment based on software containers to emulate a geographically distributed AI infrastructure. We characterize the network traffic generated by the processors when they synchronize their data, and we conduct a sensitivity analysis to evaluate how different configurations—such as the amount of data processed at once or the mathematical precision of the model—influence the communication volume and the overall training time.

The rest of this work is organized as follows.

Chapter 2 introduces the main characteristics of Large Language Models and shows the most relevant solutions that have been proposed in literature to move the process from a centralized approach towards a distributed one.

In Chapter 3, we explain the theoretical framework, detailing the decomposition of the training time and the theoretical bandwidth requirements necessary to implement decentralized AI.

Chapter 4 describes how the system model was implemented in an experimental environment, detailing the emulated topology and the software tools utilized to capture the network traffic.

In Chapter 5, we show and comment the experimental results. We performed several experiments to evaluate the impact of different batch sizes, precisions, number of connected nodes and possible bottlenecks on the overall system performance.

In conclusion, Chapter 6 summarizes the main results of this work and presents some possible future directions.

Chapter 2

Large Language Models

This chapter briefly introduces the core aspects of Large Language Models (LLMs), which are essential for understanding the performance constraints of distributed environments. In Section 2.1, key components are discussed, starting with the fundamental building block of every LLM: the Transformer Layer (2.1.1). Sections 2.1.2 and 2.1.3 explain the crucial operations performed by the model to retrieve semantic dependencies across sequences and to capture complex patterns through non-linearity.

Furthermore, Section 2.2 describes the computational lifecycle of an LLM. It details the tokenization process, namely how data is manipulated to fit the ML model (2.2.1). Subsequently, Sections 2.3 and 2.4 detail the two main computational phases of Large Language Models: training and inference. Finally, Section 2.5 reviews state-of-the-art implementations of distributed AI fabrics, while Section 2.6 outlines the primary challenges associated with this decentralized paradigm.

2.1 LLM system

An LLM system is a computational framework built on top of a Transformer architecture, designed to process every kind of possible input based on information retrieved from massive datasets [13]. It processes and generate sequential data through auto-regressive probabilistic modeling. The system consist in two main phases: training and inference. Training is a computationally intensive offline process in which the model learns from large datasets, whereas inference is a real-time execution phase during which the trained model generates responses to user requests [13]. At its core, the system functions by mapping a sequence of input tokens to a high-dimensional vector space, processing them through stacked layers of self-attention and feed-forward networks to predict the probability distribution of the subsequent token [13]. A distributed LLM system cover not only the model weights but also the distributed runtime environment required to manage the partitioning of data, the synchronization of gradients, and the storage of intermediate states such as the Key-Value (KV) cache across clusters.

2.1.1 Transformer Architecture

Today, LLMs adopt a decoder-only transformer architecture, which means that it is composed of a stack of transformer layers that predict the next token based only on the entire previous tokens [13]. Each layer has two different stages which are the Self-Attention Mechanism and the Feed-Forward Mechanism. The first one is used to capture semantic relationships between different tokens, and the second one introduces non-linearity, allowing to capture more complex pattern.

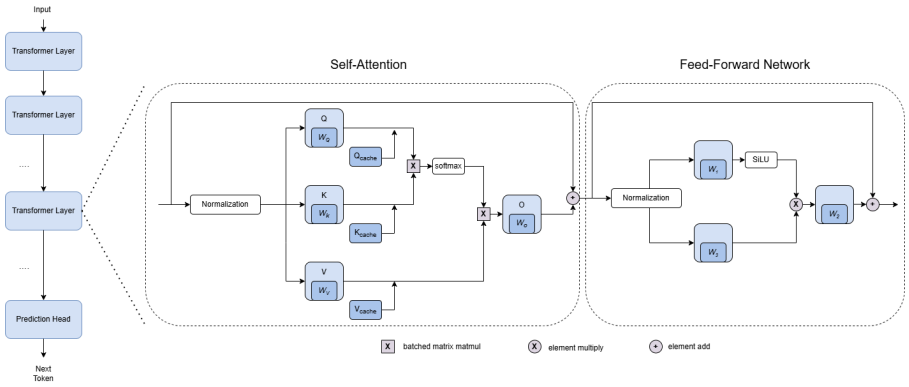


Figure 2.1: Transformer Architecture (Source [13]).

2.1.2 Self-Attention Mechanism

The self-attention mechanism is the core component that enables the model to quantify the importance of diverse tokens in the input sequence when making predictions, capturing semantic dependencies regardless of the distance between tokens. According to the formalization provided by Mitzenmacher and Shahout [13], for a given input sequence embedding $X_{pre} \in \mathbb{R}^{n \times d}$ (where n is the sequence length and d is the hidden dimension), the model applies linear transformations to project the input into three distinct matrices: the *Query* (Q_{pre}), *Key* (K_{pre}), and *Value* (V_{pre}):

$$Q_{pre} = X_{pre}W_q, \quad K_{pre} = X_{pre}W_k, \quad V_{pre} = X_{pre}W_v \tag{2.1}$$

where $W_q, W_k, W_v \in \mathbb{R}^{d \times d_k}$ are learnable weight matrices. These projections are then used to compute the attention output through the scaled dot-product formula, which includes a residual connection:

$$O_{pre} = \text{softmax}\left(\frac{Q_{pre}K_{pre}^T}{\sqrt{d_k}}\right)V_{pre} + X_{pre} \tag{2.2}$$

The softmax function ensures that the attention weights sum to one across each row, determining the relevance of each token in the context of the others.

2.1.3 Feed-Forward Networks

The output of the self-attention module is subsequently passed to the Feed-Forward Network (FFN). This component is responsible for refining the feature representation and introducing non-linearity, allowing the model to approximate more complex patterns in the data that linear attention mechanisms cannot capture.

Modern LLM architectures typically employ a specific activation function known as the Sigmoid Linear Unit (SiLU). The computation performed by the FFN layer is defined as:

$$FFN(x) = (SiLU(xW_1) \times xW_3)W_2 \tag{2.3}$$

where W_1, W_2, and W_3 represent linear transformation modules (weight matrices), and the activation function is defined as $SiLU(x) = x \cdot \sigma(x) = \frac{x}{1+e^{-x}}$ [13].

2.2 LLM Computational Processes

2.2.1 Tokenization

The tokenization process constitutes the fundamental initial step for any Large Language Model (LLM). It acts as a mathematical transformation of unstructured input data into a numerical representation readable by the model. While the nature of the input varies, ranging from massive datasets utilized during the training phase to specific user-generated prompts during inference, the core mechanism involves partitioning the input into discrete units known as tokens [13].

Tokens are granular units of data derived from decomposing larger text segments. The model processes these tokens to learn semantic relationships, enabling capabilities such as prediction, generation, and reasoning. The efficiency of this process is critical: faster token processing directly correlates with reduced latency in both learning and response generation [26].

2.3 Training

The training lifecycle involves repeating iterations, each consisting of three main phases: a forward pass to generate losses, a backward pass to compute gradients, and an optimizer step to update model weights. In distributed environments, this cycle necessitates the synchronization of gradients across thousands of accelerators, requiring the network infrastructure to provide high bandwidth and predictable latency to minimize communication overhead.

In this scenario, the Transformer layer operates within the forward pass. Unlike inference, where tokens are generated autoregressively one by one, training processes the entire input sequence (batch of tokens) in parallel. Consequently, the Key-Value (KV) cache mechanism is typically not implemented during training, as the attention mechanism computes the relationships for all tokens simultaneously rather than sequentially [13].

The process begins with the construction of a consistent and variegated vocabulary. Once the vocabulary is established, a lookup table creates a univocal relationship between raw input and integers. Subsequently, the embedding process transforms these numerical tokens into dense vectors of a specific hidden dimension, representing the peculiar dimensionality used by the hidden layers of the model to encode semantic information [10].

2.4 Inference

The inference process represents the real-time execution phase where the pre-trained model generates responses to user requests. Unlike training, the inference lifecycle is comprised of two distinct phases characterized by opposing hardware constraints: the Prefill phase and the Decode phase [13]. The entire sequence—from the initial tokenization of the input to the final de-tokenization of the output—constitutes the end-to-end request latency, representing the total response time experienced by the user [19]. The overall process is imaged in Figure 2.2.

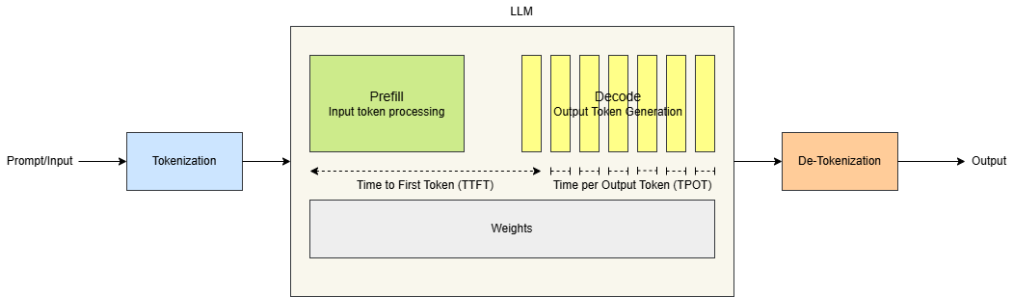


Figure 2.2: Overall tokenization process from the prompt input to the generated output (Source [19])

2.4.1 The Prefill Phase (Compute-Bound)

The Prefill phase is the initial step of LLM inference occurring immediately after the tokenization of the input prompt. In this step, the LLM processes all input tokens in parallel to retrieve context information, generating the initial Key-Value pairs and storing them in the KV cache [13]. The duration of this phase is measured as Time To First Token (TTFT) [19]. This phase is compute-bound. Since the entire prompt is available at once, the GPU can parallelize the matrix operations across the sequence length, saturating the compute units (high arithmetic intensity) rather than memory space .

2.4.2 The Decode Phase (Memory-Bound)

Once the first token is generated, the system enters the Decode phase, which operates in an autoregressive manner. The model produces subsequent tokens one by one, where each new token is conditioned on the entire history of previous tokens. The latency metric for this step is defined as Time Per Output Token (TPOT) [19]. This phase is typically

memory-bound. Because tokens are generated sequentially, the GPU must load the entire model weights and the growing KV cache from memory for every single token generation. This results in low arithmetic intensity, making the process memory-bound rather than compute-bound [13]. After generation completes, the output tokens are transformed back into readable text during the de-tokenization phase.

2.4.3 KV Cache Management

A distinct challenge in this lifecycle is the management of the Key-Value (KV) cache, which is critical for reusing intermediate computations and avoiding redundant recalculations during the decode phase [13]. However, because the cache grows dynamically with the sequence length, it creates significant memory constraints, where requests consume GPU capacity that cannot be easily freed. Consequently, preemption becomes a non-trivial issue. Interrupting a request forces a complex decision regarding its allocated KV cache: it must either be retained in GPU memory (blocking other jobs), migrated to CPU memory (incurring transfer latency), or deleted (requiring expensive recomputation upon resumption).

Therefore, defining a straightforward priority policy is difficult, as scheduling strategies must account for these memory overheads and the trade-offs between latency, cost, and cache persistence. To address this uncertainty, recent approaches propose predicting the expected response time (or output length) to inform scheduling decisions, although accurate estimation remains a complex open problem due to the variable nature of generative tasks.

2.5 Network Infrastructure

To contextually analyze the experimental results presented in later chapters, it is essential to understand the current industrial state-of-the-art. While this thesis utilizes a standard TCP/IP stack implementation (Gloo), modern high-performance infrastructure addresses distributed bottlenecks through specialized hardware and protocols designed to minimize latency and maximize throughput.

2.5.1 Scale-Across Networking: Spectrum-XGS

An AI Factory is defined as a specialized computing infrastructure designed to manage the entire AI life cycle, encompassing data ingestion, model training, fine-tuning, and high-volume inference. Unlike traditional datacenters, its primary output is intelligence itself, quantified by token throughput used to drive automated decision-making and new AI solutions [15].

As AI factories grow beyond single physical locations, traditional definitions of "Scale-Up" (within a node) and "Scale-Out" (within a cluster) are no longer sufficient. The concept of Scale-Across networking has emerged to unify geographically distributed datacenters into a single logical AI factory [16].

A critical challenge in long-haul Ethernet is the use of deep packet buffers to absorb

traffic bursts, which introduces unpredictable jitter detrimental to synchronous AI training (All-Reduce operations). NVIDIA's solution, the Spectrum-XGS platform, replaces deep buffering with distance-aware algorithms. It utilizes telemetry-based congestion control to dynamically adjust flow rates and adaptive routing to differentiate between local and long-haul traffic. This approach minimizes latency penalties and has been shown to deliver up to 1.9× higher All-Reduce bandwidth compared to standard Ethernet, enabling efficient single-job training across separated sites.

2.5.2 Transport Optimization: Kernel Bypass and Rivermax

A significant bottleneck in standard networking is the Operating System (OS) kernel overhead. In a traditional TCP/IP stack, network packets must be processed by the CPU, requiring continuous data copies and context switches between the kernel space and the user space. This mechanism inherently introduces serialization delays and unpredictable jitter, which are highly detrimental to synchronous distributed workloads.

To address this, solutions like NVIDIA Rivermax and NEIO FastSocket implement Kernel Bypass and Zero-Copy architectures [18]. By leveraging GPUDirect technology, these systems allow the Network Interface Card (NIC) to transfer data directly into the application's user-space memory or GPU memory, completely bypassing the CPU. This architecture enables dropless data transfer and ensures deterministic low latency, which is critical for real-time applications such as algorithmic trading and high-frequency inference, where even microsecond delays can degrade performance.

2.5.3 Optimized Workflow: Disaggregated Inference with Dynamo

With the rise of Reasoning Models (e.g., DeepSeek-R1) and Agentic AI, the volume of tokens generated during inference has exploded, exacerbating the resource conflict between the Prefill phase (Compute-Bound) and the Decode phase (Memory-Bound). Traditional co-location of these phases on the same GPU leads to significant resource underutilization [17].

The state-of-the-art solution is Disaggregated Serving, implemented in frameworks like NVIDIA Dynamo [17]. This architecture splits the inference process across specialized GPUs, optimizing each phase independently. The workflow is orchestrated by three core components:

- Dynamo Planner: A dynamic scheduler that monitors GPU capacity metrics and Service Level Objectives (SLOs) like Time To First Token (TTFT). It decides in real-time whether to process a request using aggregated or disaggregated resources to balance the load.
- Smart Router (KV Cache Reuse): To avoid the costly recomputation of the Key-Value (KV) cache, the Smart Router uses a Radix Tree structure to hash incoming requests. It routes prompts to specific GPU workers that already hold relevant cached contexts (e.g., system prompts or previous turns in a chat), significantly reducing computational overhead.

- NIXL (NVIDIA Inference Xfer Library): The backbone of disaggregated serving is the rapid movement of the KV cache between Prefill and Decode GPUs. NIXL provides a hardware-agnostic communication layer that abstracts the complexity of data movement across heterogeneous memory hierarchies (HBM, CPU RAM, NVMe). It manages asynchronous, point-to-point data transfers, allowing the network to act as a unified memory fabric.

2.6 Challenges and State-of-the-Art in Distributed AI

Training and serving LLMs in a decentralized manner introduces inherent challenges that vary significantly depending on the deployment scenario. As computational resource demands increase drastically, pushing the industry beyond the hardware scalability limits of a single datacenter, the AI industry is forced to move towards distributed environments to achieve optimal scalability.

2.6.1 Decentralization Paradigms

The rapid advancement of LLMs, with model scales expanding from 175 billion parameters (e.g., GPT-3) to over 660 billion (e.g., DeepSeek-R1), has dramatically increased computing resource demands [2]. To address this, the literature identifies two fundamental paradigms for decentralized LLM training. *Organizational Decentralization* involves large, well-resourced entities coordinating their large-scale facilities across different geographically dispersed datacenters, focusing on maximizing throughput and optimizing energy efficiency.

By contrast, *Community-Driven Decentralization* relies on fragmented resources pooled by individual researchers and communities. This paradigm faces severe constraints: low-bandwidth and high-latency Wide Area Networks (WANs), hardware heterogeneity, unstable resources, and strict economic limitations. Due to dynamic resource availability, robust Fault Tolerance mechanisms are critical. In complex parallel strategies like pipeline parallelism, a single node failure could trigger massive and expensive recalculations.

2.6.2 Scheduling and Fairness

Decentralized LLM efficiency requires tackling latency overhead while managing computational resources efficiently. In traditional serving systems, the First-Come-First-Served (FCFS) discipline often leads to Head-Of-Line (HOL) blocking, where short requests are stopped behind computationally intensive requests. To mitigate this, recent approaches like *LARRY (Load-Adaptive Request Reordering)* propose reordering the queue based on predicted resource consumption and current system load [8]. At the same time, ensuring fairness is critical to prevent a single client from saturating the shared cluster. The *Virtual Token Counter (VTC)* algorithm addresses this by defining a cost function based on the number of processed input and output tokens rather than raw request counts, preventing the starvation of longer requests without sacrificing overall throughput [29].

2.6.3 Prompt Sharing and Dynamic Migration

Another critical issue is redundant input data, as prompts often contain repetitive context. Architectures like *Preble* implement distributed Prompt Sharing and Prefix Caching to eliminate the need to recompute KV matrices for every request, significantly reducing the compute-bound Prefill phase [30]. Finally, to maintain training stability in decentralized and heterogeneous environments, systems like *Minder* utilize similarity-based distance checks to detect faulty nodes in real-time [1]. Concurrently, systems like *Llumnix* enable the live migration of requests and their KV cache states across model instances, reacting to unpredictable workload dynamics at runtime, ensuring service continuity, and reducing memory fragmentation [31].

Chapter 3

Theoretical Framework: The Epoch AI Model

This chapter outlines the distributed AI model adopted for this study, based on the decentralized framework proposed by Jaime Sevilla and Anton Troynikov in the following article on Epoch AI [28]. The analysis focuses on a specific Scale-Across scenario designed to address the power constraints of modern AI: a 10 GW cluster distributed across $N_{dc} = 23$ datacenters, physically located near power plants (e.g., in the US East Coast) and connected by a logical bidirectional ring with a maximum network distance $L_{max} = 4{,}800$ km.

In Section 3.1 we introduce the Distributed AI Fabric paradigm, explaining main differences with the classical centralized one, highlighting which are the possible operations and how parallelism could be managed (section 3.1.1 and 3.1.2). Section 3.3 introduces the training dynamics and time constraints required to make this geographical distribution feasible for the scenario considered. Section 3.4 decomposes the network latency components assuming standard optical infrastructure. Finally, Section 3.5 details the Ring All-Reduce algorithm governing gradient synchronization.

3.1 The Distributed AI Fabric Paradigm

An AI fabric is a highly specialized network connectivity infrastructure explicitly defined to support extreme computational demands of AI workloads, namely large-scale model training and inference. The main difference with traditional datacenter networks is that they are primarily built for general-purpose data transfer, while AI fabric functions as a true computational backplane [15]. It enables thousands of distributed hardware accelerators to operate synchronously together, transforming them logically in a single, unified device. This approach constitutes a hierarchical and multi-level parallelism. The workload distribution inherently spans three distinct networking dimensions: across multiple accelerators within a single server (Scale-Up), across multiple servers within a single datacenter (Scale-Out), and across multiple geographically dispersed datacenters (Scale-Across), as explained in Section 2.5.1 [16]. According to recent literature [14], the fundamental characteristics of a modern AI fabric include:

- Ultra-High Bandwidth and Low-Latency (Lossless Transmission): High-performance computing and Deep Learning workloads are extraordinary sensitive to latency. Even minor packet loss, forces retransmissions or causes stalls in the communication pipeline, which disturb the synchronization process between GPUs and degrades throughput. AI fabric must guarantee almost-perfect transmission, with lossless data communication.

- Optimization for Collective Operations (NCCL-Aware): An AI fabric is inherently engineered to support collective communication operations, such as All-Reduce or Broadcast, which are typically managed by the NVIDIA Collective Communication Library (NCCL) [14]. Since these operations are performed by all the processes in the group—meaning all GPUs assigned for this work—they require tight synchronization. The system can progress to the next step only when all GPUs end their operation. Consequently, the network must deliver data with minimal delay and jitter, as these factors can disrupt this timing and reduce overall efficiency.

- A robust AI fabric must seamlessly handle physical network disruptions, such as link failures or flaps, without impacting the application layer. This is crucial in the case of an unavoidable packet loss event, since a loss can de-synchronize the entire process and the network cannot converge in a deterministic and consistent manner. In default routing, BGP considers each network prefix as an independent route and, whenever a change event occurs, it individually evaluates and updates the best path for each affected prefix, leading to slower convergence in large-scale fabrics. To solve this issue, BGP Prefix Independent Convergence (PIC) is implemented, which pre-computes backup paths to enable instant recovery, regardless of the scale and size of the network, without waiting for each prefix to converge separately. This rapid recovery ensures that the synchronization between GPUs is not delayed, allowing massive AI training jobs to proceed smoothly and finish within a predictable timeframe.

- Scale-Across Connectivity: As explained in Section 2.5.1, while traditional AI fabrics connected servers strictly within a single physical datacenter (*Scale-Out*), modern infrastructures (such as the NVIDIA Spectrum-XGS platform) introduce the Scale-Across dimension. The AI fabric becomes distance-aware, utilizing telemetry-based congestion control and adaptive routing algorithms to abstract physical distances without relying on deep packet buffers which are latency-inducing.

3.1.1 Collective Communication Operations

Distributed AI fabrics rely on collective communication operations to coordinate the training or inference process across thousands of computing nodes. Unlike standard point-to-point network traffic, collective operations require the synchronous participation of all processes within a defined communication group, such as every GPUs assigned to the training job.

To manage the flow of parameters and gradients during distributed training, multiple collective operations can be performed [6]:

- Broadcast: Sends the same data from one node to all other participating nodes. In distributed training, this is typically used at initialization to ensure all computing units start with the exact same model weights.
- Scatter: Distributes distinct chunks of a larger data block from one originating node to all other nodes, providing each with a unique piece of information.
- Gather: Collects distinct data chunks from all participating nodes and merges them into a single destination node.
- All-Gather: A global extension of the Gather operation where each node sends its data to all other nodes, ensuring that every participant ultimately receives the complete, assembled set of data.
- Reduce: Aggregates data from all participating nodes using a specific mathematical operation (such as a sum, minimum, or maximum) and sends the final computed result to a single destination node.
- All-Reduce: Combines data from all nodes and distributes the aggregated result back to every node. It is used to sum the local gradients computed by each data-center and return the normalized total sum to everyone, ensuring that all distributed replicas of the model are updated identically for the subsequent step.

Collective Communication operations are visualized in Figure 3.1

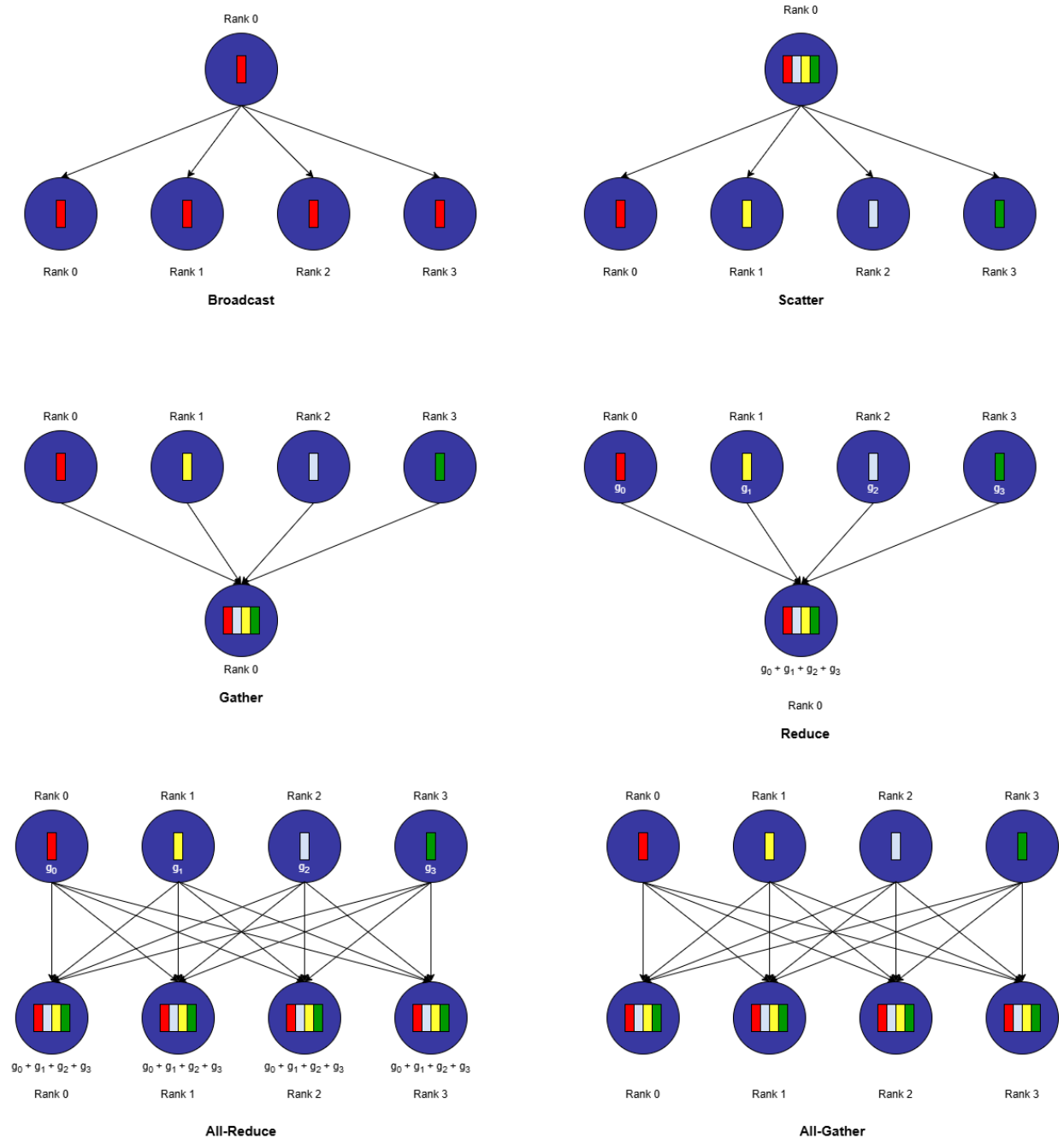


Figure 3.1: Collective Communication Operations with $ws = 4$ (Source [6])

3.1.2 LLM Parallelism

Multiple parallelisms can be performed in a LLM, addressing different specific challenges and scenarios. According to recent studies [6] [7], the main paradigms are:

- Data Parallelism: This strategy focuses on distributing data across multiple devices.

It involves replicating the entire model on each GPU, with each device processing a different portion of the input data. Gradients are computed locally and then aggregated across all devices. The aggregated gradient is used to update the model parameters on all GPUs simultaneously, ensuring they remain synchronized for subsequent steps. It is particularly useful when dealing with large datasets to speed up the training process.

- Model Parallelism: This approach is essential when a model exceeds the memory and processing capabilities of a single device. Instead of replicating the model, model parallelism splits a single large model across multiple GPUs, where different layers or parts of the model are located on different devices. Data flows sequentially through these parts. While it overcomes hardware constraints, it can be complex to implement efficiently due to the need of careful partitioning to balance the load and minimize inter-GPU communication.

- Tensor Parallelism: A specific type of model parallelism that involves dividing the model's tensors (parameters) across multiple devices. It splits individual operations or layers, such as large matrix multiplications in Transformers, across GPUs. For example, a weight matrix can be split, and partial computations are performed on different GPUs, followed by communication steps to combine the results. This method fit for large models where even individual layers are not suitable for a single device's memory.

- Pipeline Parallelism: This technique sequences the model's layers or sections across different devices, making it beneficial for models with deep architectures. To improve GPU utilization compared to naive model parallelism, the model is divided into stages, and mini-batches are divided into micro-batches. This allows different stages to process different micro-batches concurrently, reducing the issue of idle GPUs and optimizing both training and inference phases.

Different parallelism are visualized in Figure 3.2.

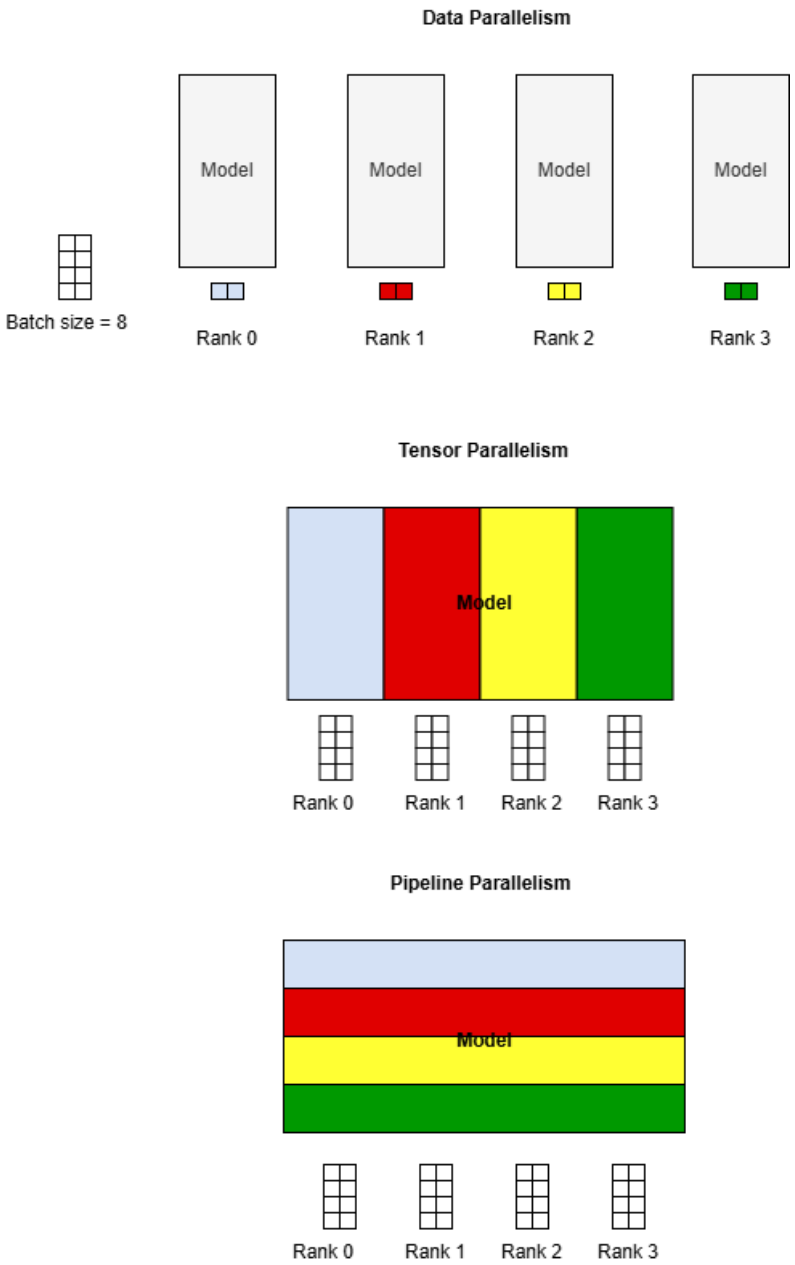


Figure 3.2: Different Parallelism discussed with $bs = 8$ and $ws = 4$ (Source [6])

3.2 Power Constraints and Cluster Dimensioning

The power demand associated with training next-generation Large Language Models (LLMs) is rapidly emerging as the primary constraint on AI scalability. Currently, leading technology companies are constructing massive centralized facilities, such as xAI's Colossus and OpenAI's Stargate, which are planned to scale between 1.2 GW and 1.5

GW [28]. However, if current scaling trends continue, AI training clusters are projected to require an overall power consumption (P_{cluster}) of up to 10 GW by the end of the decade.

Providing such a massive amount of energy in a single physical location is unfeasible. A 10 GW load exceeds the capacity of the largest individual US power plants and overwhelms local utility grids, creating stability risks. Furthermore, existing high-voltage transmission corridors are generally not rated to carry more than 6-7 GW over long distances [28]. Due to these strict infrastructural limits, this analysis considers a distributed AI fabric that leverages the spare capacity of existing power plants distributed across a wide geographical area.

To distribute this immense load, the logical cluster is composed of $N_{\text{dc}} = 23$ geographically dispersed datacenters. Therefore, the average power capacity P_{dc} required to sustain each individual datacenter within this environment is defined as:

$$P_{\text{dc}} = \frac{P_{\text{cluster}}}{N_{\text{dc}}} \tag{3.1}$$

implying $P_{\text{dc}} = 10$ GW/23=435MW.

Focusing on each datacenter individually, the computing infrastructure is built by aggregating multiple pods. Specifically, each pod is implemented with a single NVIDIA GB200 NVL72 rack, which has a nominal power consumption (P_{pod}) of 200 kW and delivers a peak throughput (TH_{pod}) of 360 PFLOPS.

To determine the total number of pods (N_{pod}) operating within the entire cluster, the overall power capacity is divided by the power consumption of a single pod:

$$N_{\text{pod}} = \frac{P_{\text{cluster}}}{P_{\text{pod}}} \tag{3.2}$$

For a 10 GW cluster, this implies a requirement of $N_{\text{pod}} = 10$ GW/200 kW = 50000 pods. Consequently, assuming an even distribution, the number of pods hosted by each individual datacenter ($N_{\text{dc}}^{\text{pod}}$) is calculated as:

$$N_{\text{dc}}^{\text{pod}} = \frac{N_{\text{pod}}}{N_{\text{dc}}} \tag{3.3}$$

yielding approximately $N_{\text{dc}}^{\text{pod}} \approx 2{,}200$ pods per datacenter.

Finally, aggregating the computational resources across the entire distributed AI fabric, the overall peak throughput of the cluster (TH_{cluster}) can be evaluated as:

$$TH_{\text{cluster}} = N_{\text{pod}} \times TH_{\text{pod}} \tag{3.4}$$

In this reference scenario, the distributed fabric achieves a massive theoretical peak throughput of $TH_{\text{cluster}} = 50000 \times 360$ PFLOPS $= 1.8 \cdot 10^{22}$ FLOPS, providing the massive computational power required for this large language model.

3.3 Decentralized Training Dynamics

In the reference scenario, the training of a large language model with $S_{model} = 72$T parameters on a dataset of $S_{data} = 2{,}000$T tokens is distributed using Data Parallelism. According to reference literature [28], this is the exact same strategy adopted by Google to split the training run of Gemini 2.5 across multiple adjacent datacenter buildings. A comparable approach was also tested by NVIDIA, which distributed the training of a 340B parameter model between two datacenters located 1,000 km apart.

The dimensions proposed in [28] exceed the computational workload of recent AI models by a significant margin. For example, the resulting computational workload represents a training run approximately 100 times larger than Grok 4, which is currently considered the largest training run to date. For further comparison, recent state-of-the-art models like Llama 4 Behemoth feature 2T parameters and were trained on over 30T tokens.

Each datacenter computes local gradients on a partition of the data, which must be synchronized globally before the next step.

3.3.1 Training Time Decomposition

The training time is the total time required to process the entire dataset and perform the gradient accumulation across all the partecipants of the training. It is the sum of all the batch time contributions during training, namely time required to train each batch size for each step. The total time to process a single batch, T_{batch}, is defined as the sum of the computation time ($T_{compute}$) and the synchronization overhead (T_{synch}):

$$T_{batch} = T_{compute} + T_{synch} \tag{3.5}$$

$T_{compute}$ correspond to the time needed to perform forward, loss, backward and optimization of the model, while T_{synch} is the time necessitated to perform the gradient synchronization until all the nodes have the same information at that step. To ensure the efficiency of the distributed run, the network overhead α_{net} must be bounded. The model imposes that synchronization should not exceed a fraction $\alpha_{net} \approx 0.05$ (5%) of the batch processing time. Thus:

$$T_{synch} \leq T_{batch} \cdot \alpha_{net} \tag{3.6}$$

The target training time is set to $T_{train} = 100$ days. This is a standard and validated timeframe for large-scale pre-training, precisely in line with the historical runs of major models such as Gemini 1.0 Ultra, Llama 3, and Grok 3. Given this timeframe and a total of $N_{batch} \approx 1.8 \cdot 10^6$ updates, the time available per batch is derived as $T_{batch} \approx 4.9$ s. Consequently, the synchronization must be completed within $T_{synch} \leq 250$ ms.

3.3.2 Critical Batch Size Scaling

To successfully hide network synchronization overhead, the computation time per training step must be sufficiently large, a condition achieved by increasing the global batch size. However, the batch size cannot be scaled arbitrarily. It is strictly bounded by a theoretical limit known as the *Critical Batch Size* (B_{crit}) [28]. Beyond this threshold, processing

additional tokens in a single batch yields significant diminishing returns in determining the optimal gradient direction, degrading the efficiency of the training process.

Critical threshold is not static, but scales proportionally with the square root of the total training dataset size (S_{data}). Consequently, this empirical law defines B_{crit} as a mathematical function of the total number of training tokens (N_{tokens}):

$$B_{crit} = k\sqrt{S_{data}} \tag{3.7}$$

In the analyzed scenario, with $S_{data} = 2 \cdot 10^{15}$ tokens, the critical batch size is approximately $1.13 \cdot 10^9$ tokens (for comparison, DeepSeek v3 was trained with a maximum batch size of 63 million tokens). This constraint fixes the number of synchronization steps.

3.4 Network Communication Modeling

The total synchronization time T_{synch} is modeled as the sum of three physical latency components: propagation delay, hop latency, and serialization delay [28].

$$T_{synch} = T_{prop} + T_{hop} + T_{ser} \tag{3.8}$$

3.4.1 Synchronization Latency Components

Propagation Delay (T_{prop})

This component depends on the physical distance L_{max} and the speed of light in the fiber medium. Standard optical fiber has a refractive index that results in a signal propagation speed of approximately $v_{fiber} \approx \frac{2}{3}c$, for a latency of $5\mu s/km$ [28].

$$T_{prop} = L_{max} \cdot 5\mu s/km \tag{3.9}$$

For the considered ring topology ($L_{max} = 4{,}800$ km), this results in a baseline propagation latency of approximately 24 ms.

Hop Latency (T_{hop})

This accounts for the processing time at network devices. For N_{hops} (corresponding to N_{dc} in the ring) and a per-hop latency τ_{hop} (typically $28\mu s$ for optical-electrical conversion in switches like Cisco NCS 2000 400 Gbps XPonder):

$$T_{hop} = N_{hops} \cdot \tau_{hop} \tag{3.10}$$

In the reference scenario, this contributes negligible delay (< 1 ms total) compared to propagation.

Serialization Delay (T_{ser})

This is the time required to transmit the gradient tensor S_{grad} onto the link with bandwidth BW_{net}:

$$T_{ser} = \frac{S_{grad}}{BW_{net}} \tag{3.11}$$

where

$$S_{grad} = N_{param} \cdot bit_{param} \tag{3.12}$$

For a 72T parameter model using 16-bit precision, $S_{grad} \approx 1.15$ Pbits ($1.15 \cdot 10^{15}$ bits). This massive data volume makes T_{ser} the dominant factor if bandwidth is insufficient.

3.4.2 Bandwidth Requirements

By rearranging the synchronization constraint 3.6 and considering synchronization latency components 3.8, the minimum required bandwidth $BW_{dc,min}$ is derived:

$$BW_{dc,min} = \frac{S_{grad}}{T_{batch} \cdot \alpha_{net} - T_{prop} - T_{hop}} \tag{3.13}$$

In the reference scenario $BW_{dc,min}$ results

$$BW_{dc,min} = \frac{72 \cdot 10^{12} \cdot 16}{4.9 \cdot 0.25 - 4800 \cdot 10^3 \cdot 5 \cdot 10^{-6} - 23 \cdot 28 \cdot 10^{-6}} \approx 5.1 \text{ Pbit/s} \tag{3.14}$$

To ensure the physical feasibility of this decentralized architecture, it is mandatory to verify that the internal computing infrastructure of each datacenter can sustain the massive WAN bandwidth requirement of $BW_{dc,min} \approx 5.1$ Pbps.

The total available network bandwidth for a single rack or pod (BW_{rack}) depends on the number of compute trays it contains (N_{trays}), the number of network ports per tray (N_{ports}), and the bandwidth capacity of each port (BW_{port}):

$$BW_{rack} = N_{trays} \cdot N_{ports} \cdot BW_{port} \tag{3.15}$$

According to the specifications of the NVIDIA GB200 NVL72 server, each rack is equipped with $N_{trays} = 18$ compute trays, each featuring $N_{ports} = 4$ ConnectX-7 ports operating at $BW_{port} = 400$ Gbps. This yields a total internal network bandwidth of $BW_{rack} = 28.8$ Tbps per rack.

Consequently, the aggregate internal network capacity of a single datacenter ($BW_{dc,internal}$), hosting approximately $N_{dc}^{pod} \approx 2{,}000$ pods, is evaluated as:

$$BW_{dc,internal} = N_{dc}^{pod} \cdot BW_{rack} \tag{3.16}$$

This configuration provides a theoretical internal capacity of $2{,}000 \times 28.8$ Tbps $= 57.6$ Pbps. Since 57.6 Pbps is an order of magnitude larger than the required WAN bandwidth ($B_{dc,min} \approx 5.1$ Pbps), the internal datacenter hardware can fully saturate the long-haul optical links without acting as a bottleneck.

3.5 Gradient Synchronization Algorithms

The system employs the *Bidirectional Ring All-Reduce* algorithm to efficiently aggregate the locally computed gradients across the N_{dc} geographically distributed datacenters [28].

To minimize idle time and hide the network latency, the gradient synchronization process is specifically designed to overlap with the computation phase. Instead of waiting for the entire backward pass to finish, modern distributed training frameworks logically partition the gradient tensors into smaller, continuous blocks. As soon as the gradients of a single block are computed, the All-Reduce process is immediately triggered over the network, while the computing units simultaneously continue the backward pass. The entire synchronization process for each batch ends only when all participants have exchanged their data and share the exact same accumulated global gradient [23].

3.5.1 The Ring All-Reduce Algorithm

The global gradient synchronization across the distributed nodes is commonly implemented using the *Ring All-Reduce* algorithm. In this topology, the participating datacenters are arranged in a logical ring, and the entire process generally consists of two distinct sequential phases: *Reduce-Scatter* and *All-Gather* [32].

- **Reduce-Scatter Phase:** The total gradient tensor is equally partitioned into distinct chunks. Over multiple communication rounds, each datacenter sends a chunk to its neighbor while simultaneously receiving a different chunk. At each step, the receiving node performs a mathematical reduction (a sum) between the incoming chunk and its own local chunk. By the end of this phase, every datacenter holds exactly one fully aggregated chunk of the global gradient.
- **All-Gather Phase:** The fully reduced chunks must now be distributed back to all participants. Over another set of communication rounds, each node transmits its completed chunk around the ring. By the end of this phase, every datacenter has received all the missing pieces, possessing the exact same, fully assembled global gradient.

This algorithm can be deployed in two equally valid variants, depending on the underlying network capabilities: the *Unidirectional* and the *Bidirectional* Ring All-Reduce [32].

The primary difference between the two lies in how the communication links are utilized. The standard Unidirectional approach propagates the data in a single direction (e.g., exclusively clockwise) across the logical ring. Conversely, the Bidirectional approach propagates the data simultaneously in both directions. In this latter setup, the data is typically split so that each node transmits data clockwise to its next neighbor and counterclockwise to its previous neighbor concurrently.

Building upon the mathematical definition of the gradient tensor size (S_{grad}) established in Equation 3.12, it is possible to analytically quantify the total volume of data exchanged across the network during a single Ring All-Reduce synchronization event (B_{event}).

Assuming a distributed topology comprising N independent nodes (World Size), the algorithm dictates that each phase strictly requires $N - 1$ communication rounds.

During any single round, the gradient tensor is logically partitioned into N equal segments, and every individual node concurrently transmits a data chunk of size $\frac{S_{grad}}{N}$ to its adjacent neighbor in the ring. Consequently, the aggregate volume of data injected

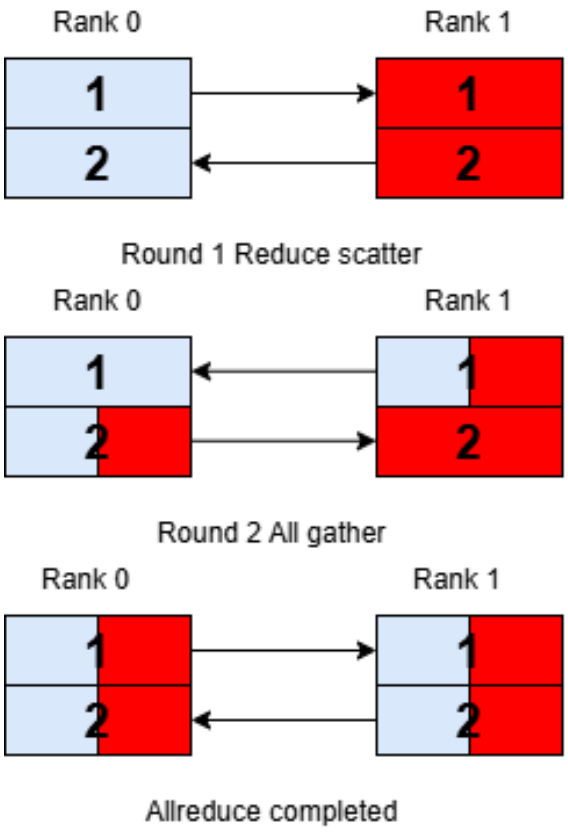


Figure 3.3: Ring All-Reduce Breakdown in Reduce-Scatter and All-Gather with $ws = 2$ (Source [32])

into the network by all N nodes simultaneously during a single round equates exactly to the full size of the gradient tensor:

$$N \times \frac{S_{grad}}{N} = S_{grad} \tag{3.17}$$

Given that the entire synchronization process consists of exactly $2(N-1)$ sequential rounds ($N-1$ for the Reduce-Scatter and $N-1$ for the All-Gather), the global network payload generated per event can be formally expressed as:

$$B_{event} = 2(N-1)S_{grad} \tag{3.18}$$

Beyond the aggregate volume of data, evaluating the efficiency of the network requires analyzing the temporal execution dynamics of the synchronization. The total time required to complete a full synchronization step (T_{ar}) can be mathematically modeled by accounting for the cumulative propagation latency (T_{prop}) and the partial serialization delay at each of the $(N_{dc}-1)$ required steps [28]:

$$T_{ar} = (N_{dc}-1)\left(T_{prop} + \frac{S_{grad}}{N_{dc} \cdot BW_{net}}\right) \tag{3.19}$$

where N_{dc} represents the number of participating datacenters (World Size), S_{grad} is the total gradient size, and BW_{net} is the available network bandwidth.

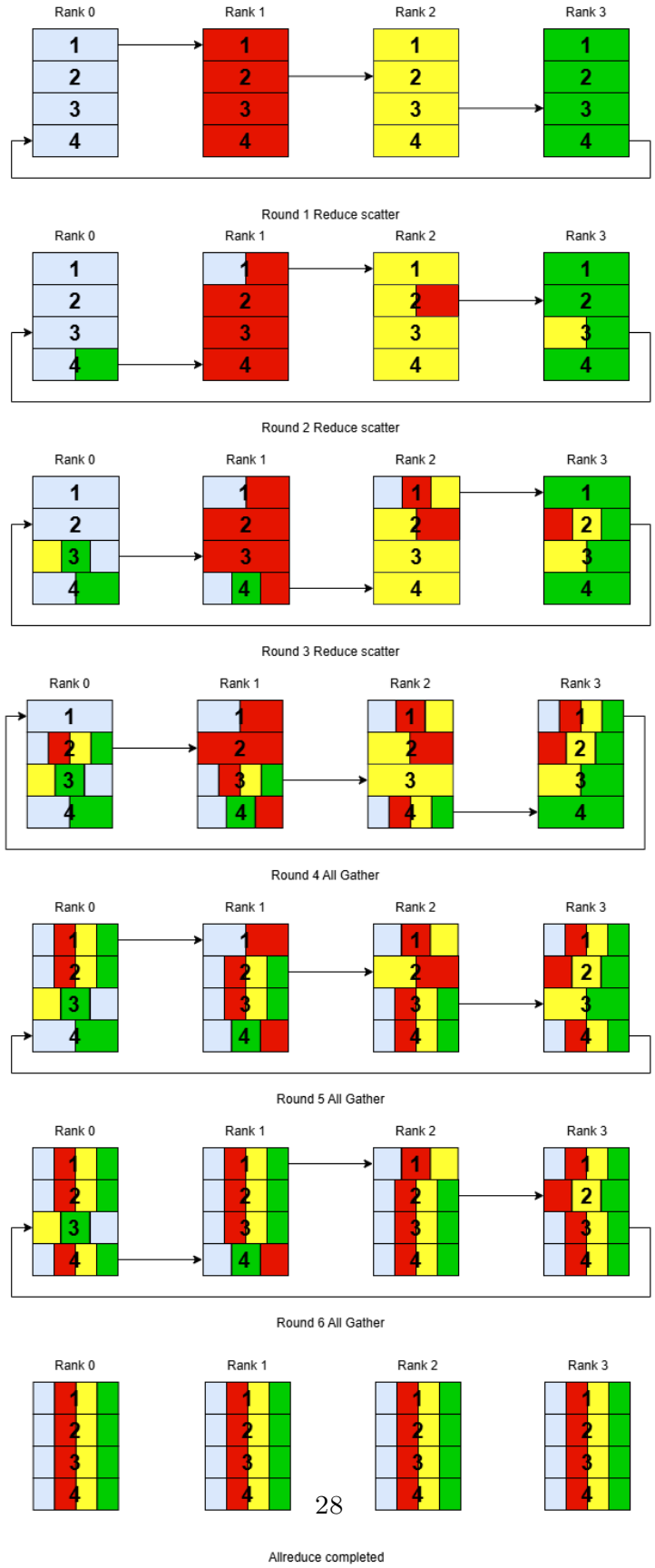


Figure 3.4: Ring All-Reduce Breakdown in Reduce-Scatter and All-Gather with $ws = 4$ (Source [32])

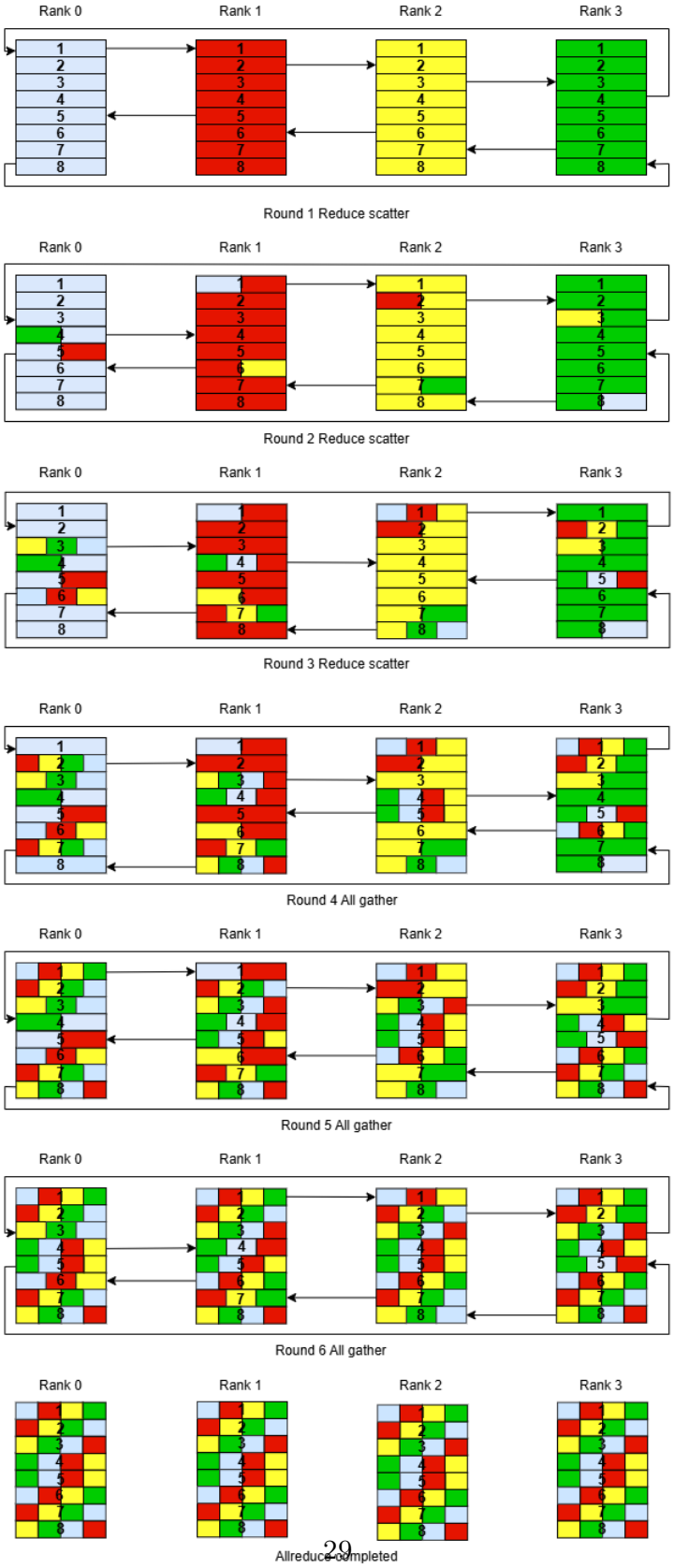


Figure 3.5: Bidirectional Ring All-Reduce Breakdown in Reduce-Scatter and All-Gather with $ws = 4$ (Source [32])

Chapter 4

Implementation and Methodology

This chapter outlines the implementation details and the methodology adopted to analyze the network traffic generated during the distributed training of Large Language Models (LLMs). Section 4.1.1 describes the experimental testbed, which utilizes software containers to emulate a decentralized network topology, while Section 4.1.2 focuses on the virtual network configuration between processes. The second part of the chapter details the software stack, focusing on the PyTorch DistributedDataParallel (DDP) framework in Section 4.2.1 and its underlying communication mechanisms in Section 4.2.2. Finally, the last part of the chapter presents the measurement methodology, including the packet-level capture techniques (Section 4.3.1) and the custom event correlation logic based on synthetic UDP markers used to evaluate the system's performance (Section 4.3.2).

4.1 Emulated Environment

4.1.1 Docker Containerization Topology

To analyze the performance of decentralized LLM training, a controlled and reproducible experimental environment was implemented using Docker containerization. The overall system architecture is illustrated in Figure 4.1.

The central hardware of the setup consists of a single physical server. On top of this physical infrastructure, the specific host environment for the experiment was deployed as a KVM, virtualized instance operating within a single-node NUMA architecture.

Specifically, the underlying system is based on an x86_64 architecture supporting both 32-bit and 64-bit CPU operating modes (with 40-bit physical and 48-bit virtual address sizes). The computational power is provided by 4 virtual CPUs (vCPUs) mapped to 4 individual sockets (configured as 1 core per socket and 1 thread per core) belonging to the Intel Xeon Processor (Cascadelake) family. To mitigate memory access latency, the processor features a hierarchical cache memory system comprising 128 KiB of Level 1 (L1) data and instruction caches, 16 MiB of Level 2 (L2) cache, and 64 MiB of Level 3 (L3) shared cache.

The virtualization layer is managed by a Kernel-based Virtual Machine (KVM) hypervisor, providing full virtualization capabilities including hardware-assisted VT-x features.

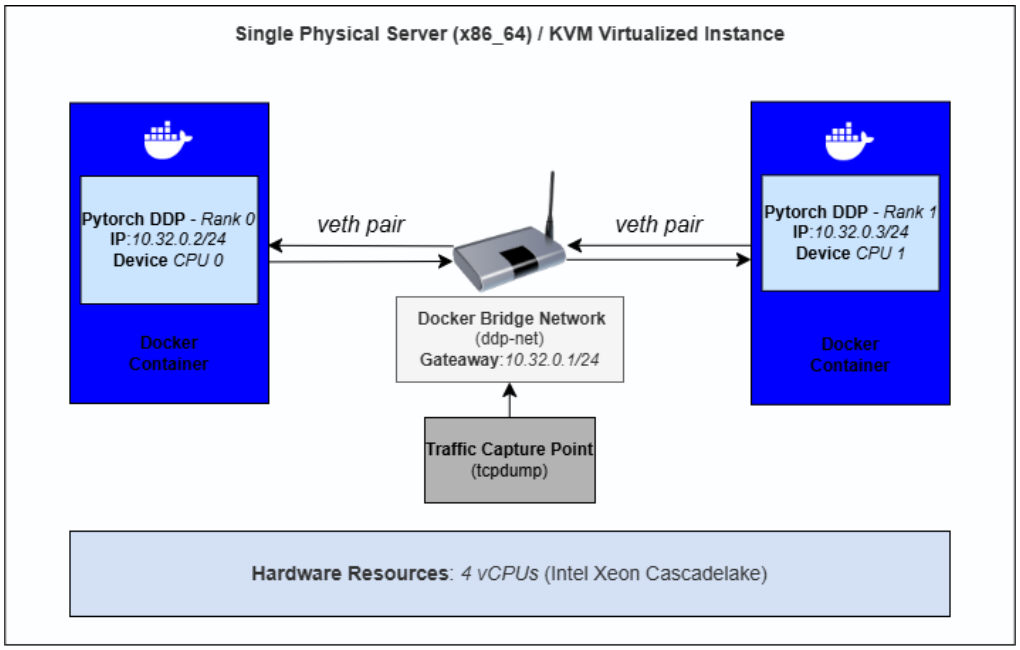


Figure 4.1: Experimental testbed architecture ($ws = 2$). The topology highlights the Docker container isolation, the CPU mapping, and the centralized network bridge serving as the traffic capture point.

Each worker type participating in the distributed architecture is built using a specific Docker image and encapsulated inside an isolated container. Utilizing Docker within this virtualized host provides significant advantages. First, it allows for the replication of a consistent emulated environment representing a small-scale decentralized AI fabric, greatly facilitating the packet-level traffic capture during training, which is the primary goal of this project. Furthermore, Docker enables strict management of the CPU resources available to each container. This level of control over the computational capacity of each node makes it possible to emulate heterogeneous hardware scenarios and accurately measure the system's performance boundaries.

4.1.2 Network Bridge Configuration

To replicate the behavior of a decentralized topology, the isolated network nodes are interconnected through a software-defined network (Figure 4.1). Specifically, the emulated topology is realized through a custom Docker bridge network named `ddp-net`. This virtual network operates on a dedicated IPv4 subnet (`10.32.0.0/24`) and is routed through a specific gateway (`10.32.0.1/24`). Each Docker container, representing a distinct node in the distributed training, is assigned an IP address within this subnet (e.g., `10.32.0.2/24`, `10.32.0.3/24`, etc.).

At the host operating system level, this configuration translates into a dedicated Linux bridge interface (e.g., `br-cecd9a809474`). The connectivity between the isolated container namespaces and this central bridge is achieved through Virtual Ethernet (`veth`)

pairs (such as `veth2496689` and `vethfd6ee1a`). This architecture forces all inter-node communication, from the initial PyTorch rendezvous phase to the Gloo-backed All-Reduce synchronization loops, to flow directly through the `br-*` interface. Consequently, this specific bridge acts as the centralized observation point, which is crucial for the measurement methodology. It allows packet analysis tools like `tcpdump` to seamlessly capture the entire bidirectional traffic flow at the kernel level without losing any transmission event.

Before the executing of the distributed training workloads, it is established the maximum theoretical throughput of the containerized infrastructure. To measure the baseline capacity of the virtual Linux bridge (named `ddp-net`), a dedicated network stress test was conducted utilizing the `iperf3` utility.

The test was orchestrated by deploying Docker containers attached to the same virtual network. To thoroughly saturate the link and accurately emulate the concurrent traffic patterns typical of distributed applications, the `iperf3` client was configured to generate 4 parallel TCP streams (`-P 4`) in reverse mode (`-R`) over a sustained duration of 30 seconds (`-t 30`).

The measurement recorded a maximum baseline link capacity of approximately 25 Gbit/s between the containers. This extremely high throughput analytically confirms that the virtual Docker bridge does not introduce arbitrary bandwidth bottlenecks. Consequently, any network congestion or throughput degradation observed during the subsequent Distributed Data Parallel (DDP) experiments can be strictly attributed to the intrinsic burstiness, serialization overhead, and algorithmic structure of the gradient synchronization process, rather than the underlying physical link.

4.2 Software Stack

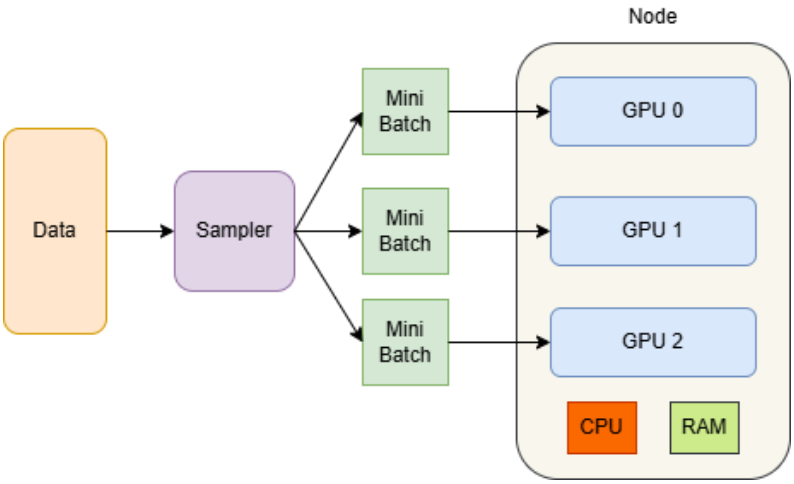
4.2.1 PyTorch DistributedDataParallel (DDP)

The distributed behavior of the system is implemented using the PyTorch

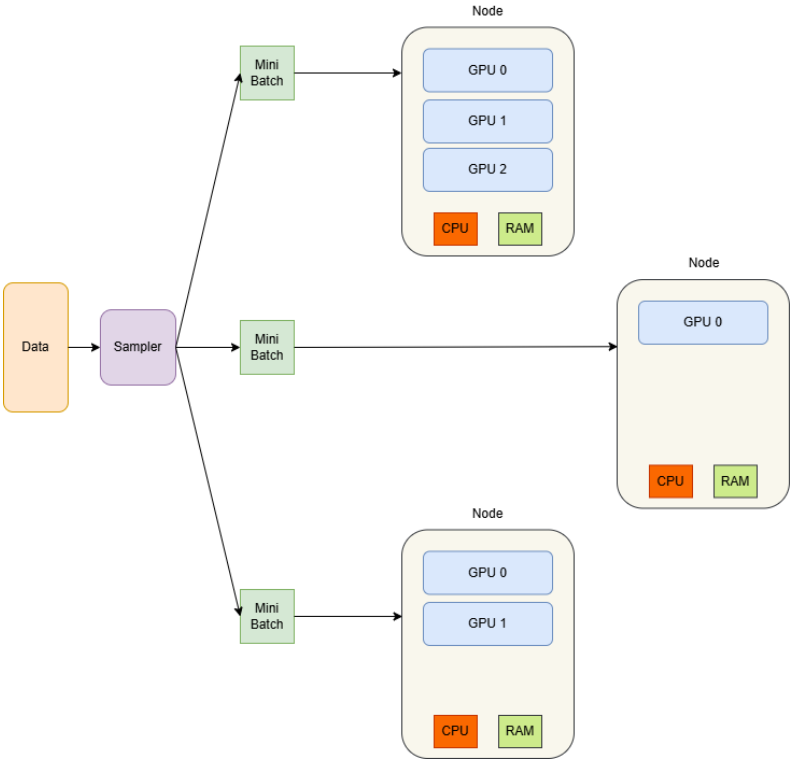
`DistributedDataParallel` (DDP) module. As detailed in the official PyTorch documentation [21] [22] [23] and supported by reference literature [6], DDP provides synchronous distributed training by transparently managing data parallelism across multiple network-connected nodes. Unlike the older `DataParallel` (DP) approach, DDP employs a multi-process architecture. By spawning a separate, independent Python interpreter for each participating rank, DDP prevents synchronization bottlenecks and is the recommended standard for distributed training. Two different approaches are visualized in Figure 4.2.

The fundamental limitation of `DataParallel` is its single-process, multi-threaded architecture. Although DP can utilize multiple GPUs, it operates exclusively within a single machine and is severely bottlenecked by Python's Global Interpreter Lock (GIL) [6].

The GIL ensures that only one thread can execute Python bytecode at a given time, even on multi-core processors, causing significant GIL-thrashing and extra interpreter overhead. Conversely, DDP spawns a separate and independent process for each GPU. This multi-process approach entirely bypasses the GIL, avoids CUDA stream contention, and enables each process to have dedicated and reliable access to its assigned GPU, making it faster even in single-node environments. Furthermore, in the DP paradigm,



(a) Data Parallel Architecture



(b) Distributed Data Parallel Architecture

Figure 4.2: Comparison between Data Parallel and Distributed Data Parallel architectures (Source [6])

the model is replicated across multiple GPUs, but a single central process is responsible for gathering and aggregating all the computed gradients. This centralized aggregation becomes a severe performance bottleneck and scales highly inefficiently as the number of devices increases [6].

DDP, on the other hand, eliminates the central aggregation node entirely. It utilizes highly optimized collective communication primitives to synchronize gradients over the network. During the backward pass, DDP uses autograd hooks—callback functions triggered by PyTorch's automatic differentiation engine (Autograd) as soon as a specific gradient is computed—registered for each parameter to automatically trigger the gradient synchronization across all processes [21]. This ensures that gradients are perfectly averaged and kept in sync distributedly, without relying on a centralized bottleneck.

Another crucial advantage of DDP over DP lies in how the optimizer step is handled. In DP, after the central node aggregates the gradients and updates the model, the newly computed parameters must be explicitly broadcasted back to all GPU replicas, creating massive data transfer overhead. In DDP, because the All-Reduce operation guarantees that every process holds the exact same averaged gradients at the end of the backward pass, each process independently maintains its own local optimizer and performs a complete optimization step. Since all model replicas start from the identical initial state and apply the identical gradient updates at every iteration, they remain perfectly synchronized implicitly. Consequently, no broadcast step is needed after the backward pass, drastically reducing the time spent transferring massive tensors between nodes and maximizing overall training throughput.

In the DDP paradigm, the computational workload is distributed across multiple independent entities called nodes, which represent the physical or virtual machines involved in the cluster [6]. Within each node, the execution is carried out by distinct processes, where each process acts as an independent worker and is strictly mapped to a single computational device to avoid synchronization bottlenecks. To manage the communication and data exchange among all these independent workers, PyTorch requires the creation of a logical structure known as a process group [22]. The total number of processes participating globally in this process group is defined as the world size [6]. To identify each participant uniquely across the entire distributed job, the system assigns a global integer identifier called global rank, which ranges from zero to the world size minus one. In multi-node architectures, a distinction is made between this global rank and a local rank, which identifies a process related to a single physical machine [6]. For instance, in a cluster of two nodes with four GPUs each, the global ranks range from zero to seven, while the local ranks on each node reset and range from zero to three. However, in the containerized implementation adopted for this experimental setup, each Docker container acts as an isolated node hosting exactly one training process mapped to its dedicated CPU resources. Because there is a strict one-to-one mapping between nodes and processes, the local rank is constantly zero for every container. Consequently, this architectural distinction becomes operationally irrelevant, and the system relies solely on the global rank to orchestrate the distributed job. Finally, by convention, the process associated with global rank zero is designated as the master process; this specific rank acts as the central coordinator responsible for managing the initial rendezvous phase,

establishing the network connections, and broadcasting the initial model weights to ensure all workers are perfectly aligned before any training iteration commences. In Figure 4.6 the overall training evolution is visualized, considering the baseline scenario with two ranks.

Initialization and Process Group

As reported in the official documentation [22], the DDP lifecycle begins with the creation of a process group (utilizing the Gloo backend) to establish the collective communication context. Within this multi-process architecture, each worker operates as an independent entity identified by a unique Process ID (PID) and is allocated memory space, ensuring strict computational isolation from the other processes. The process group provides the logical structure defining the participating entities, the communication rules, and the available collective primitives. Upon initialization, each new process retrieves its configuration from specific environment variables (namely `MASTER_ADDR`, `MASTER_PORT`, `WORLD_SIZE`, and `RANK`). By convention, the process assigned to Rank 0 acts as the central coordinator for the initial exchange, functioning as the rendezvous endpoint. During this rendezvous phase, all participating ranks connect to the master node via TCP at the specified address and port. They interact with a distributed store hosted on Rank 0, explicitly communicating their metadata to confirm their participation. Specifically, each process registers the following information:

- Its assigned global rank identity (`RANK`).
- Its readiness state to join the cluster.
- The selected communication backend (e.g., Gloo).
- The expected total number of participants (`WORLD_SIZE`).

The master process (Rank 0) actively monitors the process and blocks the execution until exactly `WORLD_SIZE` processes have successfully checked in. Only when all expected workers have arrived and mutually verified their parameters, the rendezvous is considered complete, and the distributed environment is officially established for the training loop. In Figure 4.3 is visualized the process group of baseline scenario discussed in Section 5.2 with $ws = 2$.

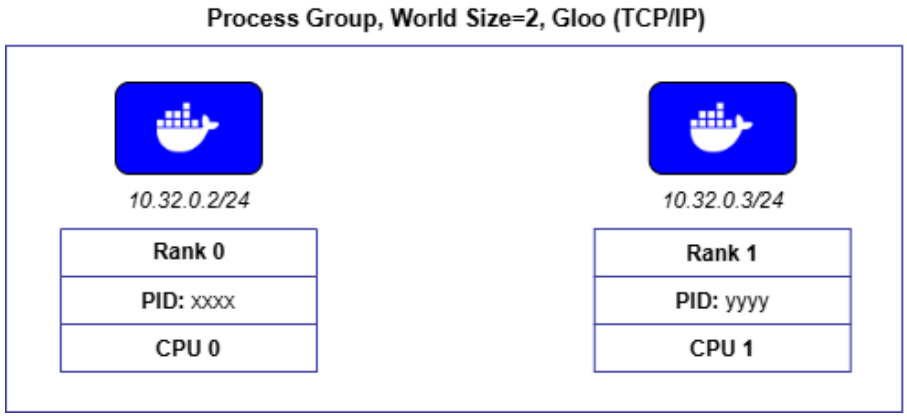


Figure 4.3: Process Group of baseline scenario with $ws = 2$

Model Architecture

Within the containerized environment, each rank independently instantiates the model architecture and downloads its corresponding pre-trained weights. However, to guarantee that all distributed replicas are perfectly aligned before the training loop begins, the DDP constructor automatically broadcasts the initial model state from the master process (Rank 0) to all other participating ranks [21]. To optimize the bandwidth utilization and the overall efficiency of this collective broadcast operation, the underlying communication relies on a logical N-layer tree topology, which systematically distributes the payload across the cluster without bottlenecking the master node [3]. Figure 4.4 illustrates the model loading on both nodes of the baseline scenario.

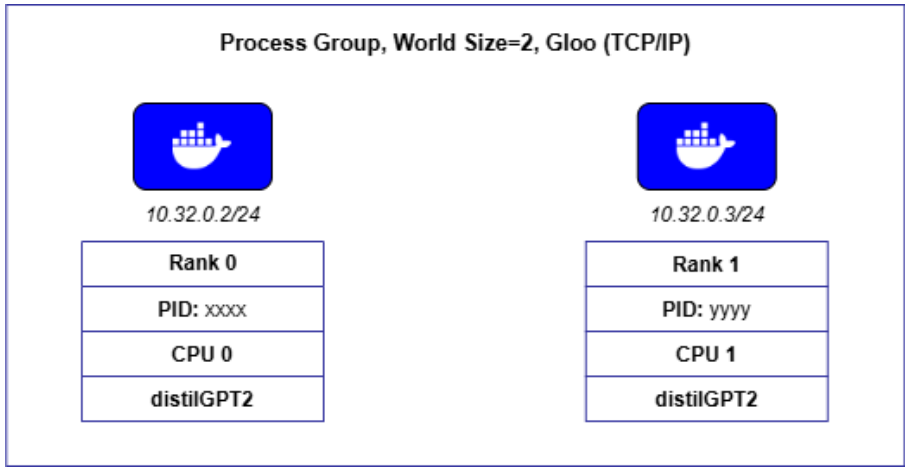


Figure 4.4: Model Distribution of baseline scenario with $ws = 2$

Data Partitioning

Within the distributed environment, each rank independently loads a local copy of the dataset. To ensure that each process computes on a unique segment of the training data, a `DistributedSampler` is utilized [25]. This sampler dynamically shards the dataset by generating a deterministic, non-overlapping partition of indices for each rank based on its unique global identifier. Consequently, this guarantees that the local mini-batches are completely distinct across all ranks, allowing the training computation to proceed in parallel without any data duplication or overlap. In Figure 4.5 is dispayed this process for baseline scenario.

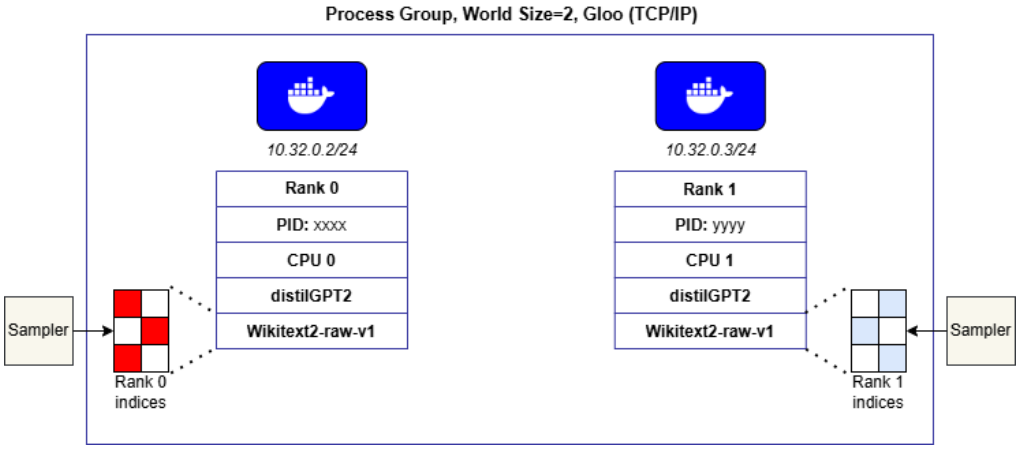


Figure 4.5: Dataset Sharding using DataLoader in baseline scenario with $ws = 2$

Forward Pass and Loss Computation

During the forward pass, each DDP process passes its unique data partition to the local model replica. The forward propagation and the subsequent loss computation are executed entirely locally on each rank, without requiring any inter-process communication over the network.

Backward Pass

The most critical mechanism of DistributedDataParallel occurs during the backward pass. After the local loss is computed, the backward propagation traverses the computational graph in reverse order, calculating the gradient of the loss with respect to each individual parameter to determine how weights must be updated. As these local gradients are evaluated, they must be synchronized across all ranks. However, synchronizing every single gradient individually as soon as it is computed would result in severe inefficiency. For instance, a single 768×768 weight matrix in the DistilGPT2 architecture requires approximately 2.25 MB in FP32, meaning that a parameter-by-parameter synchronization would lead to thousands of micro-communications and massive network overhead. For this reason DDP implements a technique known as Gradient Bucketing [21]. Instead of

transmitting each parameter individually, gradients are grouped into larger, contiguous memory blocks before synchronization, a mechanism that will be detailed in Section 4.2.3.

All-Reduce and Optimizer Step

As these asynchronous collective operations execute across the network, the underlying communication backend natively performs a pure mathematical sum of the local gradients from all participating nodes [21]. Once this network-level summation is completed, the DistributedDataParallel module automatically scales the aggregated result by dividing it by the total world size (ws) [6]. This produces a globally averaged gradient, mathematically expressed as

$$g = \frac{1}{N} \sum_{r=1}^{N} g_r$$

Upon the completion of the entire backward pass, all buckets have been successfully synchronized, meaning that every rank strictly holds the exact same averaged gradient values in its memory. Because all distributed replicas originate from an identical mathematical state, the subsequent local optimizer step applies these symmetric updates independently but deterministically. This fundamental mechanism ensures that the model parameters remain perfectly synchronized across the entire distributed cluster without requiring any explicit weight broadcasting after the backward pass, thereby maximizing overall training throughput.

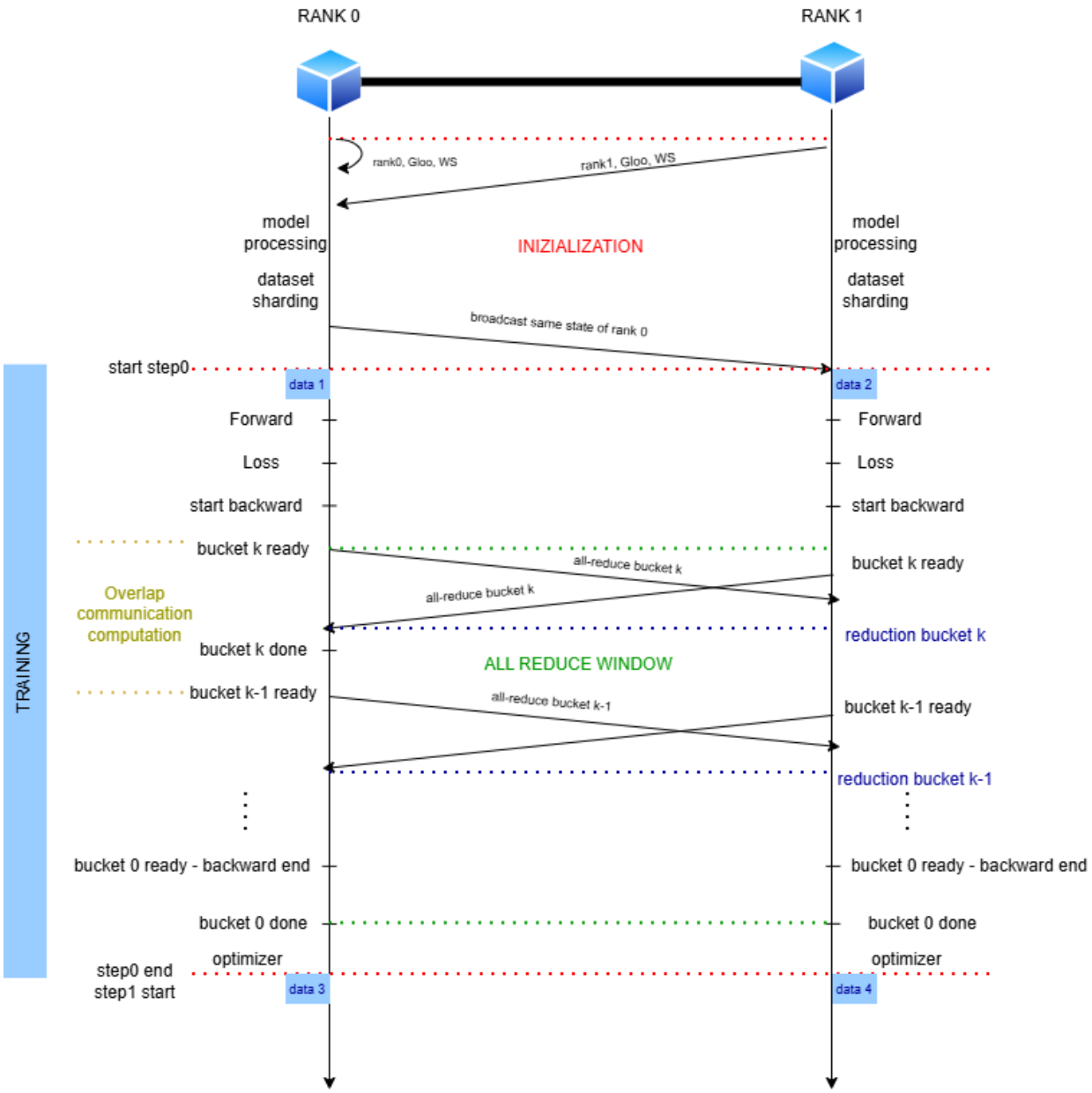


Figure 4.6: Overall training process with $ws = 2$

4.2.2 Gloo Backend and TCP/IP Transport

To physically execute the collective communication primitives required for gradient synchronization, the logical process group relies on an underlying network communication backend. In this experimental containerized setup, the Gloo backend is explicitly deployed to manage the data exchange between the isolated workers [22]. Developed as a platform-agnostic communication library, Gloo is the official recommended standard for

distributed training environments operating on CPU hosts. Unlike hardware-specific libraries such as NCCL, which are strictly optimized for GPU interconnects, Gloo natively utilizes the standard TCP/IP protocol stack to route data packets over the network. By establishing TCP sockets between the participating nodes, this backend efficiently orchestrates all the critical collective operations, including the initial parameter Broadcast and the asynchronous All-Reduce of the gradient buckets, ensuring a reliable transmission across the emulated distributed fabric.

4.2.3 Gradient Bucketing Implementation

To mitigate the communication bottleneck, the framework employs a Gradient Bucketing mechanism managed by an internal Reducer [21]. During the model construction phase, the Reducer allocates the model parameters into continuous memory buckets up to a configurable target size, which defaults to 25 MB [23]. Furthermore, this allocation is performed in the reverse order of the model's parameters, mathematically anticipating the exact sequence in which gradients will become ready during the reverse traversal of the backward pass.

Moreover, at construction time, DDP registers specific autograd hooks for every parameter in the model [21]. As the backward pass proceeds layer by layer, whenever a local gradient is fully computed, its corresponding autograd hook fires, explicitly marking that specific gradient as ready for reduction. Once all the individual gradients assigned to a particular bucket transition to the ready state, the Reducer does not wait for the entire backward pass to finish. Instead, it immediately and automatically triggers an asynchronous All-Reduce collective operation for that specific bucket across all participating ranks, seamlessly overlapping the network communication of the current bucket with the ongoing gradient computation of the subsequent layers.

4.3 Measurement Methodology

4.3.1 Traffic Capture (tcpdump/Wireshark)

To systematically characterize the network traffic at the packet level, continuous captures were performed directly on the virtual Linux bridge interface. Monitoring the bridge is particularly advantageous in this containerized topology, as it acts as the central transit point for all inter-node communications, ensuring a complete and centralized view of the exchanged data.

By employing the `tcpdump` utility, the raw network packets routed between the isolated containers were recorded, capturing the entire communication lifecycle ranging from the initial TCP three-way handshakes to the rendezvous phase and the subsequent parameter broadcast. To optimize storage and focus exclusively on routing metadata rather than the actual tensor payloads, a snapshot length parameter (`-s 128`) was applied, truncating each packet to preserve only the essential headers, such as source addresses, destination addresses, and packet lengths. Furthermore, the `-nn` flag was utilized to disable DNS name resolution, thereby minimizing processing overhead during the capture.

Each recording session was executed over the duration of an entire training epoch before being manually terminated. Moreover, network offloading features on the bridge interface—specifically Generic Receive Offload (GRO), Generic Segmentation Offload (GSO), and TCP Segmentation Offload (TSO)—were explicitly disabled before the experiments. Disabling these features prevents the operating system from artificially grouping packets together before they are captured. This ensures a lossless and packet-by-packet recording of the actual physical network traffic.

Finally, the resulting packet capture (PCAP) files were exported and subsequently analyzed using Wireshark to extract precise flow statistics and evaluate the overall impact of the emulated network constraints on the distributed training loop.

4.3.2 UDP Marker Injection for Event Correlation

Correlating the physical network-layer packets captured by `tcpdump` with the logical, application-layer execution phases of the PyTorch training loop represents a significant methodological challenge. Because the DistributedDataParallel module asynchronously overlaps local gradient computation with network transmission, relying solely on host-side application timestamps fails to accurately reflect the exact wire-level communication windows.

To bridge this observability gap and achieve precise packet-to-step correlation, a custom network synchronization logic based on synthetic User Datagram Protocol (UDP) markers was engineered. A marker serves as a deterministic, lightweight network bookmark injected directly into the captured traffic trace. To implement this mechanism, the system leverages the `register_comm_hook` API provided by the DDP framework [23].

This communication hook is natively designed to intercept the internal Reducer's operations, allowing custom execution precisely when a gradient bucket becomes ready for synchronization. By injecting state-tracking logic into this hook, the system actively monitors the sequence of gradient buckets in real-time. It detects the processing of the first bucket to mark the exact initiation of the All-Reduce window, and it tracks the end of the final bucket to identify the exact microsecond the gradient synchronization for a given training step concludes. At these precise boundaries, the master process (Rank 0) explicitly sends a synthetic UDP datagram into the network interface.

To ensure these markers do not interfere with the TCP-based Gloo synchronization traffic, they are exclusively routed to a dedicated, unused destination port (port 9999). Furthermore, to guarantee granular traceability during post-processing, the current training step index is mathematically encoded directly into the UDP source port using the formula $src\_port = 40000 + step$. When the resulting packet capture (PCAP) files are subsequently analyzed in Wireshark, these UDP datagrams function as exact, packet-aligned bounding boxes. This architectural instrumentation successfully isolates the physical All-Reduce communication windows for each individual training iteration, enabling a rigorous, artifact-free calculation of the instantaneous network throughput and the per-step synchronization latency.

4.3.3 Metrics Definition

To rigorously evaluate the interaction between local computation and network transmission, specific performance metrics are defined. The physical network parameters measured are:

- *Bytes per Event ($Bytes_{AR}$)*: The total volume of data physically routed across the network interface during a single gradient synchronization burst.
- *Event Duration ($Dur/event$)*: The exact elapsed time on the wire, measured exclusively between the initial and final UDP markers bounding that specific synchronization window.
- *Peak Throughput (TH_{peak})*: The instantaneous network speed achieved during the active transmission phase. It is calculated for each training step using the formula:

$$TH_{peak} = \frac{Bytes_{AR} \cdot 8}{Dur/event} \tag{4.1}$$

The parameters retrieved at software-level are:

- *Step Duration (T_{step})*: The total wall-clock time required by the application to complete a full training iteration.
- *All-Reduce Duration (T_{ar})*: The wall-clock time measured by the PyTorch framework to complete the gradient synchronization phase. Unlike the physical *Dur/event*, this software-level metric incorporates system overheads, memory bucketing preparation, and wait times between nodes.
- *Communication Overhead (OH_{ar})*: The percentage of the step time spent by the framework executing the All-Reduce operation.

Chapter 5

Validation of the Model

This chapter presents the empirical validation of the decentralized Artificial Intelligence fabric model discussed in the previous sections. Section 5.1 details the experimental environment, including the specific characteristics of the DistilGPT2 architecture (Section 5.1.1) and the partitioned Wikitext-2 dataset (Section 5.1.2) utilized to generate the network workloads. The first phase of the experimental analysis is presented in Section 5.2, which establishes a baseline performance using a two-node configuration (World Size = 2). This section rigorously investigates the underlying synchronization dynamics, the impact of batch size scaling, and the exact network penalty introduced by higher mathematical precision. Finally, Section 5.3 expands the cluster to four nodes (World Size = 4) to empirically validate the theoretical communication overhead, throughput degradation, and latency spikes induced by the Ring All-Reduce algorithm over the emulated network.

5.1 Experimental Setup

The experimental environment is strictly designed to emulate a decentralized AI fabric utilizing a containerized infrastructure. Multiple independent processes are orchestrated to execute synchronous distributed training, relying on physical network metrics, to rigorously evaluate the interaction between local computation and network communication. To systematically characterize the system under various operational regimes, the following foundational parameters are explicitly defined and modulated throughout the experiments:

- **Model:** The reference architecture utilized to generate the network workload is DistilGPT2, a simplified version of GPT2.
- **Dataset:** The training relies on the Wikitext-2 collection. To ensure manageable epoch durations and successfully isolate the network synchronization phases without requiring prohibitive computational times, the training dataset is explicitly reduced to a deterministic subset of exactly 400 samples.
- **World Size (ws):** This topological parameter dictates the scale of the distributed fabric, representing the total number of independent ranks (containerized processes)

participating in the collective training. The experimental baseline operates with a minimal configuration of $ws = 2$ and it is subsequently scaled to $ws = 4$.

- **Local Batch Size (bs):** This defines the exact number of dataset samples processed locally by an individual rank during a single training step. Modulating the batch size is crucial, as it directly alters the ratio between the local computation volume and the overall frequency of network synchronizations required per epoch.
- **Tensor Encoding (Precision):** This parameter specifies the mathematical representation format of the model's weights and gradients. The baseline utilizes standard 32-bit floating-point (FP32) precision, which is subsequently compared against 64-bit double precision (FP64) to mathematically evaluate the network penalty introduced by doubling the physical payload size over the wire.
- **Sequence Length:** To standardize the computational intensity of the forward and backward passes across all nodes, the sequence length is strictly fixed at 64 tokens for every processed sequence.

5.1.1 Model: DistilGPT2

To ensure a manageable computational load while generating a sufficient volume of gradient synchronization events, the DistilGPT2 model [5] was selected as the reference architecture (Figure 5.1). Developed by Hugging Face, DistilGPT2 is an English-language Transformer-based model created utilizing a model compression technique known as knowledge distillation [27].It was explicitly designed to be a faster, lighter, and more memory-efficient version of the standard 124-million-parameter GPT2 model [10], trained to reproduce the behavior of its larger "teacher" model while requiring significantly fewer computational resources.

The architecture comprises 6 identical Transformer blocks and features a hidden embedding dimension (H) of 768. Overall, the model contains precisely 81,912,576 parameters. When the model weights and gradients are mathematically represented in standard 32-bit floating-point (FP32) precision, this translates to a structural memory footprint of approximately 328 MB (81.9×10^6 parameters $\times$ 4 bytes).

In particular, consistent with the adopted Data Parallelism paradigm, the physical model architecture is not partitioned across the network. Instead, ahead of the initiation of the training loop, the complete DistilGPT2 model is explicitly downloaded from Internet and instantiated independently within the local memory of every single containerized process (rank). This preliminary initialization guarantees that all participating nodes begin the training phase possessing an exact, identical replica of the pre-trained model before any distributed gradient synchronization takes place.

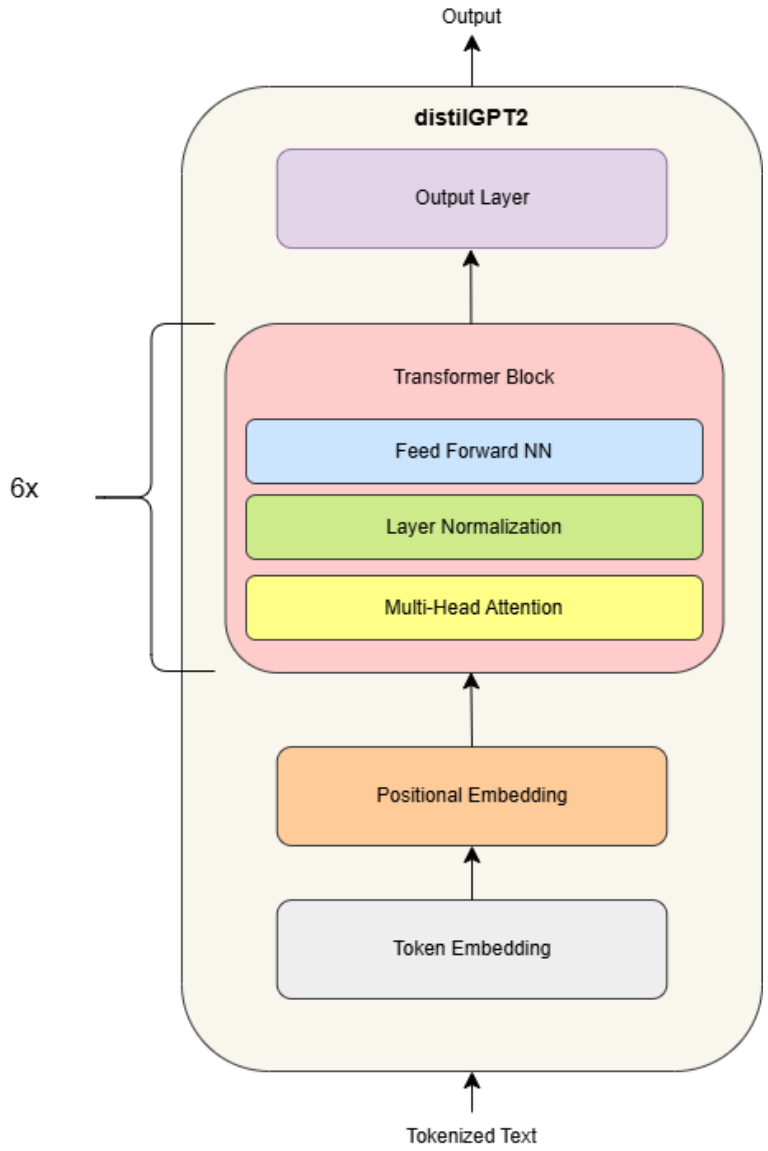


Figure 5.1: distilGPT2 model breakdown (Source [10])

Tokenization and Vocabulary

Following the exact same pre-processing procedure of GPT2 model, DistilGPT2 operates on a fixed vocabulary of exactly 50,257 tokens, generated through the Byte-Pair Encoding (BPE) algorithm [4]. Unlike traditional word-level tokenizers that may struggle with out-of-vocabulary words, the BPE algorithm initializes a fundamental base vocabulary of 256 byte-level characters. This mathematical formulation ensures that any arbitrary text sequence, including unknown words or special characters, can be constructed from these base bytes without ever encountering unknown ([UNK]) tokens. During its initial creation, the algorithm iteratively scans a massive training corpus to identify and merge

the most frequent adjacent pairs of symbols into new, longer subword tokens, stopping only when the target vocabulary limit of 50,257 is reached. Once the vocabulary is fully built, each resulting subword is deterministically mapped to a unique discrete integer index. It is crucial to note that this vocabulary generation is performed only once during the tokenizer's pre-training phase. Consequently, when processing new, unseen samples from an independent dataset the text is systematically mapped to integer tensors relying entirely on this pre-computed, static vocabulary structure, ensuring a consistent embedding lookup without any further tokenizer updates.

Embeddings and Tensor Dimensionality

Once the raw text is tokenized, the data is aggregated into batches to feed the model [11]. At the very entrance of the neural network, the input is mathematically structured as a two-dimensional matrix of size $[B, L]$, where B represents the Batch Size (the number of independent samples processed simultaneously) and L is the Sequence Length (the number of discrete tokens per sample). Every single element in this $[B, L]$ matrix is initially just an integer. The first critical transformation occurs in the embedding layer: this layer acts as a dense lookup table that maps every single token into a continuous numerical vector of hidden dimension $H = 768$. Physically, this means that for each individual token, an array of 768 floating-point values is generated and inserted into the data structure, expanding the initial 2D matrix into a three-dimensional tensor of shape $[B, L, H]$.

However, because the Transformer architecture processes all L tokens of the sequence strictly in parallel rather than sequentially, it lacks any inherent concept of word order. To inject spatial awareness, a Positional Encoding vector [10], computed using deterministic sinusoidal functions, is element-wise added to each 768-dimensional word embedding.

Transformer Blocks and Attention

The $[B, L, H]$ tensor systematically flows through the 6 identical Transformer blocks of the DistilGPT2 architecture. The core computational component of each block is the Multi-Head Causal Self-Attention mechanism [9]. This mechanism dynamically evaluates the semantic relevance between tokens by projecting the $[B, L, H]$ input into distinct Query (Q), Key (K), and Value (V) weight matrices (Section 2.1.2). Because DistilGPT2 is an autoregressive generative model, a causal mask is applied to the raw attention scores, mathematically setting future token values to negative infinity to strictly prevent the model from accessing subsequent words during next-token prediction.

The computation is distributed across 12 parallel "heads". Instead of a single macroscopic attention computation, the multi-head approach splits the hidden dimension into smaller representation subspaces (specifically, $768/12 = 64$ dimensions per head) executed concurrently, enabling the model to simultaneously attend to different syntactic and contextual relationships. The outputs of all 12 heads are then concatenated, passed through a Feed-Forward Neural Network (FFN) equipped with a GELU activation function, and stabilized via residual connections as explained in Section 2.1.3. While these internal operations are mathematically complex, their fundamental architectural property

is dimensional preservation.

The data tensor enters and exits every single Transformer block strictly as $[B, L, H]$. This exact dimensional consistency across the entire model is precisely what makes the memory footprint, the gradient sizes, and the resulting network synchronization payload perfectly deterministic and predictable during distributed training.

5.1.2 Dataset: Wikitext-2-raw-v1

To generate the textual workload required for the distributed training experiments, the *Wikitext-2-raw-v1* dataset was selected [12]. Derived entirely from selected Wikipedia articles, this standard corpus is widely utilized in Natural Language Processing (NLP) to evaluate and train generative language models.

Structure and Characteristics

Unlike other datasets that present shuffled, isolated, or heavily pre-processed sentences, the *raw* version of Wikitext-2 specifically preserves the original paragraph and article structure. This property is particularly ideal for autoregressive models like DistilGPT2, as it allows the network to learn long-term semantic dependencies across coherent, contiguous text passages. The raw text includes structural formatting such as headings, subheadings, and complex syntactic structures. For instance, the initial sequential entries in the dataset include strings such as *" = Valkyria Chronicles III = "* followed by related contextual paragraphs like *"The game began development in 2010 , carrying over..."*.

Dimensionality and Experimental Reduction

The complete, unmodified training split of the Wikitext-2 dataset contains exactly 36,718 distinct text passages, which corresponds to a total memory footprint of approximately 18.26 MB for the raw text structures. However, executing the training loop over the entire corpus would result in excessively long epochs, heavily diluting the network-level metrics under prolonged computational phases. Therefore, to ensure a manageable computational load and to cleanly isolate the network synchronization bursts (the All-Reduce windows) without requiring prohibitive runtimes, the dataset was intentionally and deterministically reduced to a subset of exactly first 400 samples for the baseline experimental runs.

Distributed Sampling and Sharding

In the implemented Data Parallelism architecture, the physical division of data does not occur prior to the execution of the training script. Instead, every independent process (rank) begins by loading a complete, identical copy of the entire 400-sample dataset directly into its local container memory. It is only after this full dataset initialization that the data is logically partitioned to maximize the efficiency of the distributed paradigm.

To achieve this mathematically, a `DistributedSampler` is applied independently within each container. Relying on the `RANK` and `WORLD_SIZE` environment variables inherently injected into the containers, the sampler precisely identifies the total scale of the

cluster and the specific identity of the local process.

In particular, the sampler is explicitly configured with the `shuffle=True` parameter. At the beginning of every training epoch, the global list of dataset indices is randomly shuffled. Immediately after this randomization, the sequence of indices is evenly sharded across the ws participating nodes. For instance, in a baseline $ws = 2$ configuration, although both ranks possess the entirety of the physical dataset in memory, Rank 0 will exclusively draw its local batches from every alternate index (starting from 0) of the newly shuffled indices, while Rank 1 will be restricted to drawing from the remaining ones [25].

Because of this active shuffling mechanism, the data distribution is deliberately non-deterministic across different epochs. Nevertheless, within any single epoch, this dynamic sharding guarantees a strictly non-overlapping logical partition of the input data. This prevents any data duplication during the concurrent forward passes while ensuring that the global model collaboratively explores the entire dataset exactly once per epoch.

5.2 Experimental Analysis: World Size = 2

The initial validation baseline evaluates a minimal distributed setup comprising two nodes (World Size = 2). This configuration establishes the foundational network traffic patterns for gradient synchronization prior to scaling the cluster. The topology is visualized in Figure 5.2

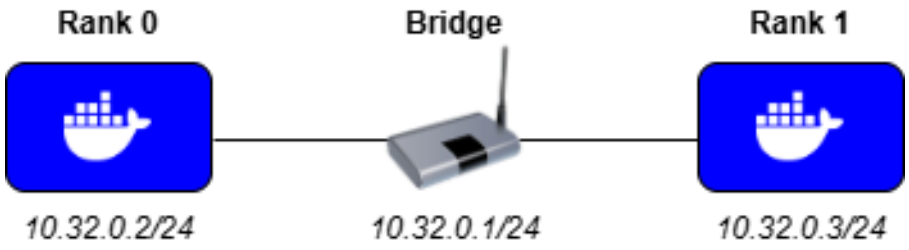


Figure 5.2: Topology of baseline scenario with $ws = 2$

5.2.1 Synchronization Analysis

Traffic Burstiness and System Resource Utilization

An analysis of the instantaneous network throughput combined with system resource profiling reveals that the distributed training process operates in a strictly compute-bound regime characterized by severe traffic burstiness. Unlike continuous data streams, the Distributed Data Parallel (DDP) paradigm generates discrete, highly concentrated spikes of network activity separated by relatively long periods of network silence as can be seen in Figure 5.3.

The behavior of the distributed training process is intrinsically tied to the algorithmic nature of the Distributed Data Parallel (DDP) paradigm. During the forward pass, the

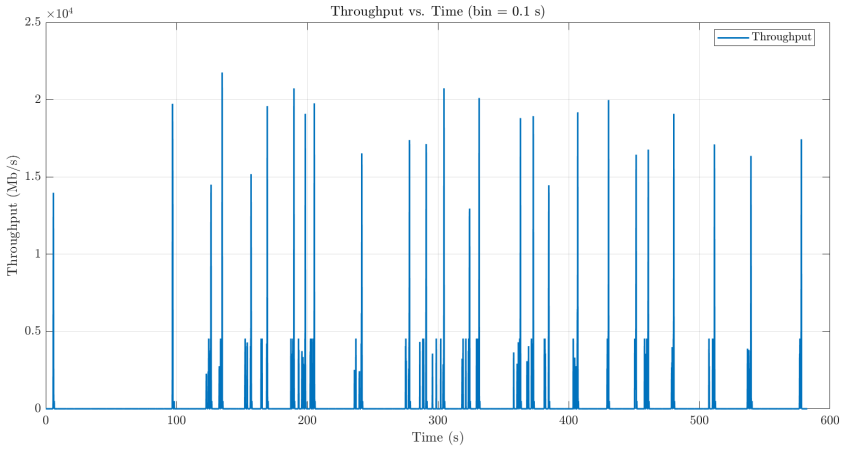


Figure 5.3: Throughput over time of the traffic captured on the bridge during training($ws = 2, bs = 8, FP32$)

loss calculation, and the initial phases of the backward propagation, the system operates in a strictly compute-bound regime where the computational units are fully saturated while the network remains completely idle.

Profiling the containerized environment during these compute-intensive phases, by sampling Docker statistics at one-second intervals throughout the entire training epoch, reveals an extremely high CPU utilization. As illustrated in Figure 5.4, after the initial setup phase, the CPU load maintains a constant average of approximately 200% per rank. This effectively saturates all four available CPU cores of the underlying server, seamlessly distributed across the two processes. Minor fluctuations in individual rank utilization are dynamically balanced by the host operating system scheduler, which redirects available computational cycles to the most demanding process to ensure maximum overall efficiency.

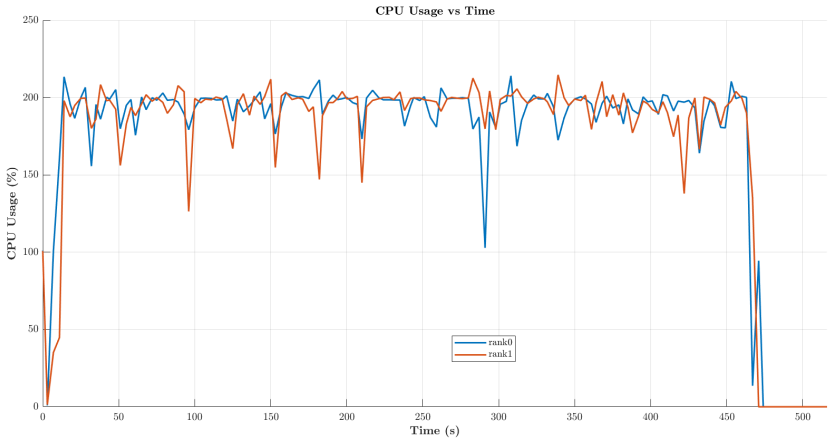


Figure 5.4: CPU utilization over time for each rank during a baseline training epoch ($ws = 2, bs = 8, FP32$).

Once the initialization phase concludes, the system enters the training loop. As highlighted by the network profiles, the DDP backend does not wait for the entire backward pass to finish before initiating network communication. Instead, gradients are logically grouped into smaller, continuous buckets. As soon as a bucket is populated and ready, its respective All-Reduce communication is triggered over the bridge, effectively overlapping with the ongoing backward computation of the remaining layers. This pipelined architecture significantly increases the overall training performance by successfully hiding network latency behind computational tasks.

This underlying mechanism dictates the traffic behavior. As illustrated in Figure 5.5, the cumulative network traffic exhibits a strict, step-like pattern. Almost the entirety of the payload exchanged across the virtual bridge is confined strictly within the gradient synchronization windows. Consequently, the instantaneous throughput experiences severe, physiological micro-bursts precisely synchronized with the algorithmic all-reduce phases, as shown in Figure 5.6.

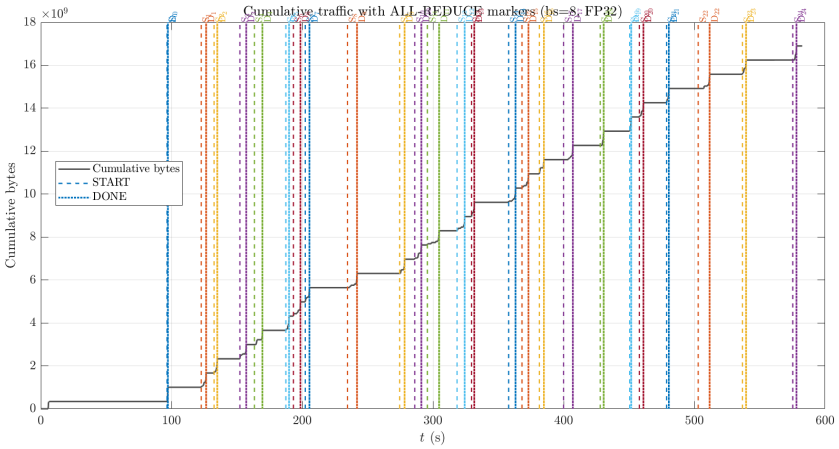


Figure 5.5: Cumulative network traffic over a single epoch ($ws = 2, bs = 8, FP32$).

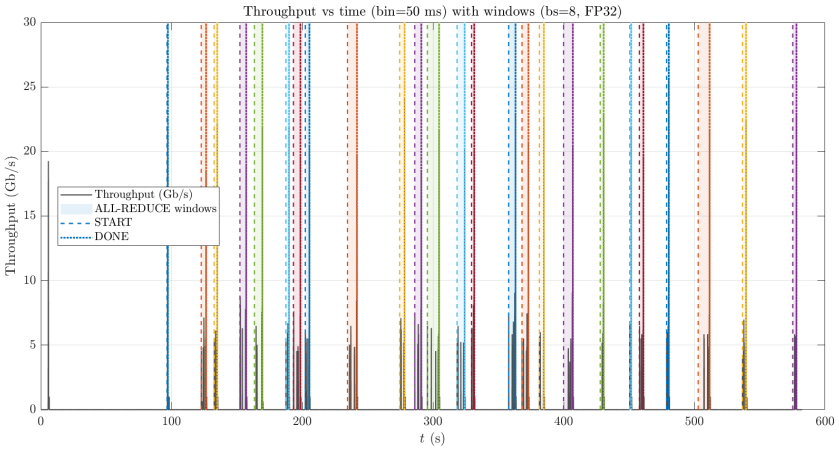


Figure 5.6: Instantaneous throughput per All-Reduce event.

By sampling the cumulative Network TX and RX metrics of the two containers, as illustrated in Figure 5.7, this discrepancy becomes evident: Rank 0 transmits an initial payload that Rank 1 receives, creating an offset. It is worth noting that the slight initial surge in the cumulative traffic, occurring prior to the first formal training step, corresponds exactly to the initial model broadcast from Rank 0 to Rank 1, analyzed in the previous section. Excluding this preliminary initialization overhead, the bidirectional transmissions remain perfectly symmetric throughout the entire epoch.

The total cumulative traffic of approximately 8.2 GB transmitted by each rank corresponds exactly to the theoretical expectation. According to the Formula 3.12, the data transmitted by each individual rank per step is equal to $\frac{2(N-1)S_{grad}}{N}$. For the baseline scenario ($N = 2$), with a gradient size $S_{grad} \approx 328$ MB and an epoch consisting of

$N_{steps} = 25$ steps, the total transmission per rank is calculated as:

$$B_{TX,event} = \frac{2(N-1)S_{grad}}{N} \times N_{steps} = \frac{2(2-1) \times 328}{2} \times 25 = 328 \times 25 = 8200 \text{ MB} \approx 8.2 \text{ GB} \tag{5.1}$$

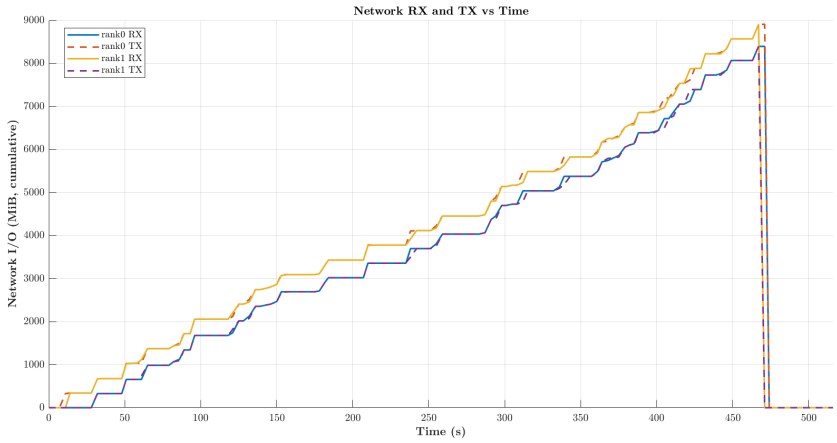


Figure 5.7: Comparison of cumulative transmitted (TX) and received (RX) traffic across ranks, highlighting the initial model broadcast discrepancy.

Regarding memory utilization, Figure 5.8 shows a stable footprint of roughly 2.8 GiB allocated per rank. This quantity is significantly higher than the mere 328 MB payload exchanged over the network during a gradient synchronization event. This discrepancy is due to the complex memory breakdown required by the training loop. While the model parameters and their corresponding gradient tensors require exactly 328 MB each, the Adam optimizer maintains two additional state variables [24](momentum and variance) per parameter, consuming an extra 656 MB of fixed structural memory [6]. Finally, the remaining memory footprint is dynamically allocated during the step by the forward activations, the Autograd computational graph, and the communication buffers instantiated by the DDP backend, bringing the total memory to achieve this huge reservation [6].

It is also important to note that the actual memory actively utilized by the process intrinsically oscillates throughout the training epoch. While parameters, gradients, and optimizer states constitute a persistent memory baseline, forward activations, intermediate Autograd tensors, and communication buffers are dynamically allocated and released during every single training step. Consequently, the actual memory occupied by the tensors exhibits fluctuations, rising during the forward pass and dropping after the backward pass and optimizer step [20].

Barrier vs. No-Barrier Impact

To analyze the temporal dynamics of the distributed training loop, an experiment was conducted to evaluate the impact of explicit network synchronization. This scenario compares the baseline training execution against a modified execution that invokes the

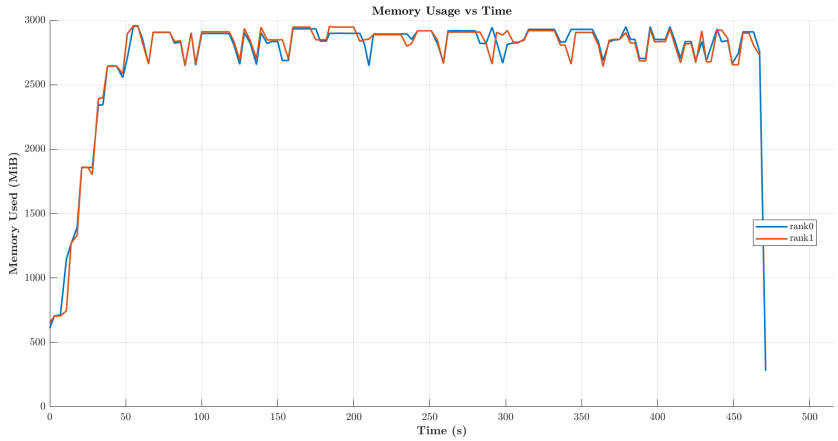


Figure 5.8: Memory consumption over time for each rank ($ws = 2, bs = 8, FP32$).

`torch.distributed.barrier()` primitive at the end of each training step. Under normal conditions, PyTorch's Distributed Data Parallel (DDP) maximizes throughput by overlapping mathematical computation with network communication.

Introducing an explicit barrier blocks all processes until the entire cluster reaches the synchronization point, forcing a rigid serialization between the computational and network phases. As expected, this explicit synchronization does not alter the transmitted payload size. The global metrics in Table 5.1 and the cumulative traffic trends in Figure 5.9 confirm that both executions exchange the same structural volume across 25 All-Reduce events. The per-event payload distribution in Table 5.2 and Figure 5.10 shows an identical payload of approximately 656 MB.

As documented in Table 5.3, the median step duration (T_{step}) increases from 20296 ms in the baseline scenario to 21491 ms when the barrier is applied. Analyzing the network packets in Table **??**, the median duration of the communication burst elongates from 3.732 s to 4.066 s. This artificial delay reduces the network efficiency. As illustrated in Table 5.2 and Figure 5.11, the median peak throughput (TH_{peak}) drops from 1406.9 Mbps to 1291.3 Mbps, while the maximum peak throughput drops from 8392.9 Mbps to 4919.6 Mbps. Despite the overall step time increasing, the communication overhead (OH_{ar}) slightly decreases from 27.8% to 26.4%. This occurs because the barrier's idle wait times are absorbed into the overall step duration.

Global Epoch Metrics	**No-Barrier**	**Barrier**	**Ratio (B/NB)**
Number of All-Reduce Events	25	25	1.00
Total bytes [MB]	16899.5	16735.7	0.99
Total duration [s]	582.5	608.0	1.04
$TH_{avg,overall}$ [Gbps]	0.232	0.220	0.95

Table 5.1: Overall network metrics for the complete training epoch comparing No-Barrier and Barrier executions.

Per-Event Statistics	**No-Barrier**	**Barrier**	**Ratio (B/NB)**
Bytes/event (min) [MB]	656.248	656.245	1.00
Bytes/event (median) [MB]	656.271	656.280	1.00
Bytes/event (max) [MB]	656.296	656.298	1.00
Dur/event (median) [s]	3.732	4.066	1.09
Dur/event (max) [s]	8.961	12.694	1.42
TH_{peak} (median) [Mbps]	1406.9	1291.3	0.92
TH_{peak} (max) [Mbps]	8392.9	4919.6	0.59

Table 5.2: Statistical breakdown of the 25 individual All-Reduce events comparing No-Barrier and Barrier executions.

Wall-Clock Metrics	**No-Barrier**	**Barrier**	**Ratio (B/NB)**
T_{step} (median) [ms]	20296	21491	1.06
T_{step} (max) [ms]	91730	73023	0.80
T_{ar} (median) [ms]	4621	4222	0.91
T_{ar} (max) [ms]	8932	12653	1.42
OH_{ar} (median) [%]	27.8	26.4	0.95
OH_{ar} (p75) [%]	35.5	35.6	1.00

Table 5.3: Wall-clock temporal metrics and communication overhead comparing No-Barrier and Barrier executions.

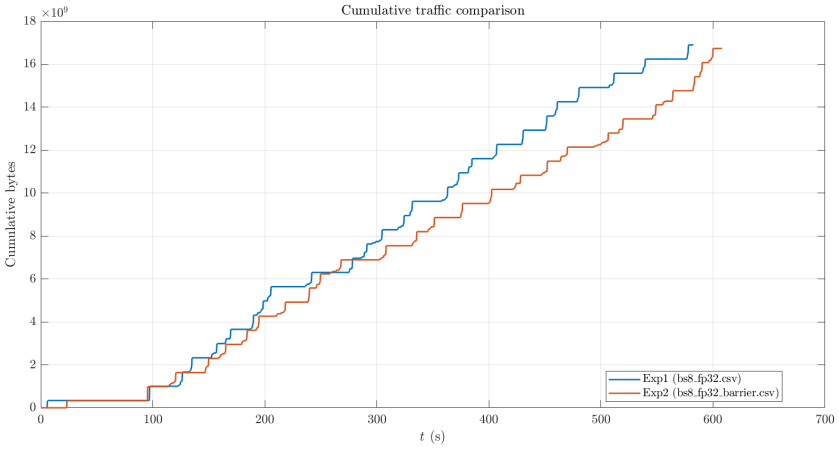


Figure 5.9: Cumulative network traffic comparison between No-Barrier and Barrier scenarios.

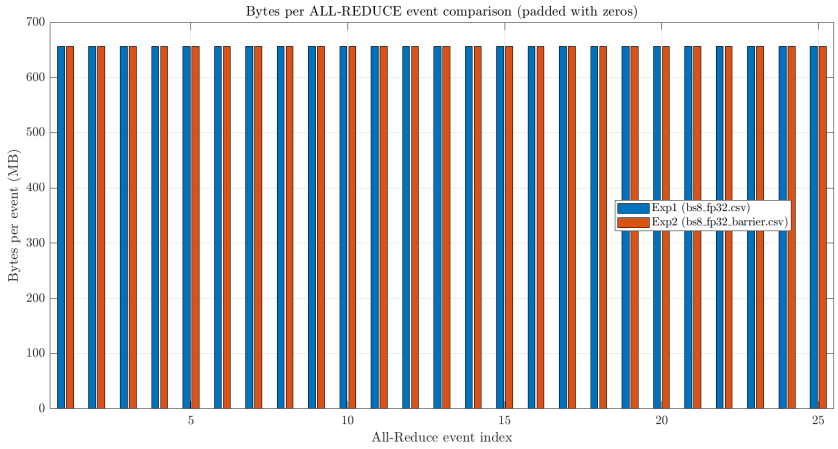


Figure 5.10: Bytes exchanged per individual All-Reduce event. The explicit synchronization primitive does not alter the structural payload.

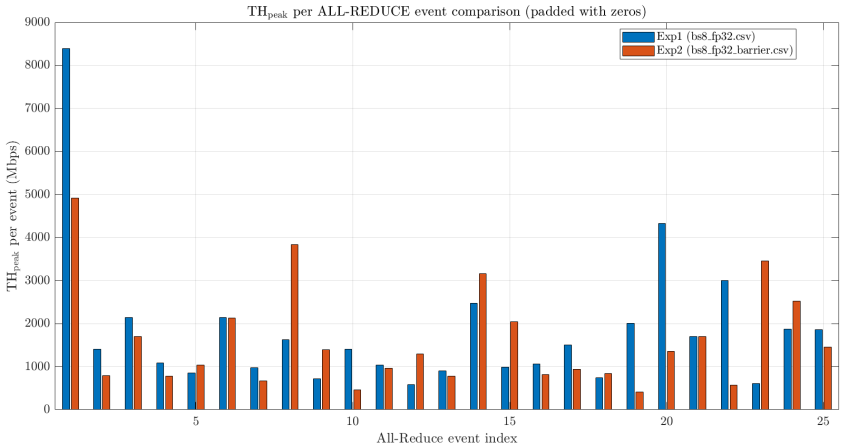


Figure 5.11: Peak throughput (TH_{peak}) per event. The barrier forces idle wait times, significantly degrading the instantaneous burst efficiency.

Validation of Flow Symmetry

The symmetrical behavior of the distributed system, initially observed in the Section 5.2.1 through the Docker resource profiling of each individual rank, must be empirically demonstrated at the network level. To definitively validate the structural symmetry of the Distributed Data Parallel (DDP) implementation, the overall network traffic captured during the baseline scenario ($bs = 8$, FP32) was logically decomposed and analyzed across its two independent directional flows Rank 0 $\rightarrow$ Rank 1 and Rank 1 $\rightarrow$ Rank 0 (10.32.0.2/24 $\rightarrow$ 10.32.0.3/24 and 10.32.0.3/24 $\rightarrow$ 10.32.0.2/24).

Mathematical Expectations For a cluster of $N = 2$ nodes, the theoretical volume of data exchanged during a single Ring All-Reduce event is $B_{event} = 2(N-1)S_{grad} = 2S_{grad}$. Given the 328 MB gradient size of the DistilGPT2 model, the total payload over the virtual bridge per synchronization step should sum to approximately 656 MB. Viewed directionally, the expectation is a perfectly symmetrical exchange: the flow from Rank 0 to Rank 1 and the flow from Rank 1 to Rank 0 must each transfer exactly $S_{grad} \approx 328$ MB per step over an identical time window (T_{ar}), leading to an identical instantaneous peak throughput (TH_{peak}).

Flow Analysis To empirically validate this symmetry, the full packet capture of the baseline epoch was decomposed into its two constituent directional flows. As reported in Table 5.4 and visually captured in the cumulative plot in Figure 5.12, the network traffic exhibits a slight asymmetry: Rank 0 transmits a total of 8695.5 MB, while Rank 1 transmits 8204.0 MB.

Global Epoch Metrics	**02 → 03**	**03 → 02**	**Ratio (03/02)**
Number of All-Reduce Events	25	25	1.00
Total bytes [MB]	8695.5	8204.0	0.94
Total duration [s]	582.5	582.5	1.00
$TH_{avg,overall}$ [Mbps]	119.4	112.7	0.94

Table 5.4: Overall epoch metrics capturing the directional flows for the baseline scenario ($bs = 8$, FP32).

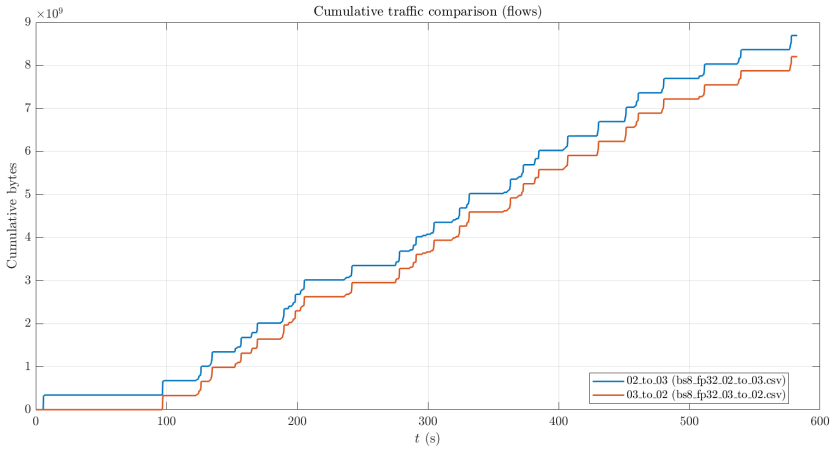


Figure 5.12: Cumulative traffic comparison between the two directional flows over a complete training epoch.

This overall discrepancy (a ratio of 0.94) is strictly tied to the initialization analyzed in the Section 5.2.1. The vertical step observed at the very beginning of the 02 → 03 curve represents the model broadcast. Once the actual training loop begins, the two

curves grow with perfectly parallel step-like increments, indicating a strictly symmetric behavior.

Event Symmetry A granular analysis of the 25 individual All-Reduce synchronization windows conclusively proves the algorithmic symmetry of the system. Figure 5.13 isolates the exact payload exchanged directionally during each event. The distribution is mathematically flat: both flows consistently transfer a median of precisely ≈ 328.13 MB per step, perfectly adhering to the S_{grad} theoretical expectation.

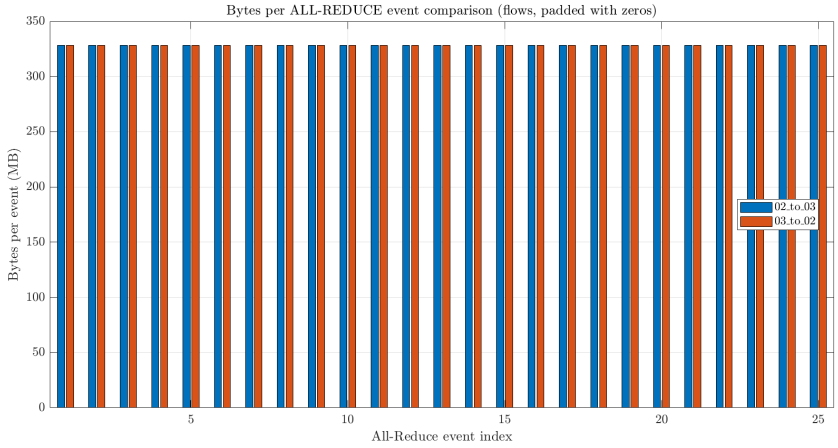


Figure 5.13: Bytes transferred per individual All-Reduce event for both directional flows.

Because the Ring All-Reduce algorithm in a $ws = 2$ setup forces a rigid synchronization, the duration of the event is universally dictated by the slowest link. Consequently, as detailed in Table 5.5 and confirmed by Figure 5.14, the calculated peak throughput (TH_{peak}) is absolutely identical between the $02 \rightarrow 03$ and $03 \rightarrow 02$ flows for any given step index.

Per-Event Statistics	**02 → 03**	**03 → 02**	**Ratio (03/02)**
Bytes/event (min) [MB]	328.112	328.125	1.00
Bytes/event (median) [MB]	328.138	328.134	1.00
Bytes/event (max) [MB]	328.156	328.145	1.00
Dur/event (median) [s]	3.732	3.732	1.00
Dur/event (max) [s]	8.961	8.961	1.00
TH_{peak} (median) [Mbps]	703.5	703.5	1.00
TH_{peak} (max) [Mbps]	4196.3	4196.6	1.00

Table 5.5: Statistical breakdown of the 25 individual All-Reduce events per flow.

While the throughput exhibits high volatility across different steps (ranging from medians of 703.5 Mbps up to intense micro-bursts of nearly 4196 Mbps) due to variations

in local computation overlap, the bidirectional symmetry remains mathematically intact. This physical validation guarantees that the network load is evenly distributed across the links during the training phase, confirming the structural reliability of the setup.

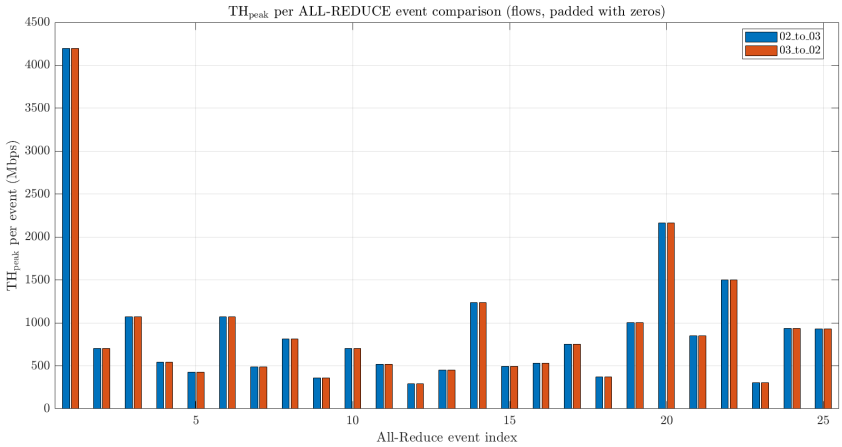


Figure 5.14: Peak throughput (TH_{peak}) recorded for each independent All-Reduce event. The values are perfectly mirrored across both flows.

5.2.2 Batch Size Sensitivity Analysis: $bs = 8$ vs $bs = 16$

To evaluate how local computation scaling impacts the network synchronization workload, a comparative sensitivity analysis was conducted between two distinct configurations: a baseline local batch size of 8 and an increased batch size of 16.

Mathematical Expectations and Step Frequency In the Distributed Data Parallel (DDP) paradigm, the global batch size (bs_{glob}) processed by the cluster at each step is the product of the local batch size and the world size.

$$bs_{glob} = bs_{local} \times ws$$

Given the fixed dataset size of exactly $N_{samples} = 400$, the number of training steps, and consequently, the number of strictly required All-Reduce synchronization events, scales inversely with the global batch size.

For the baseline scenario ($bs_{local} = 8$), the cluster processes 16 samples per step, resulting in exactly 25 All-Reduce events per epoch:

$$N_{step,bs=8} = \left\lfloor \frac{400}{8 \times 2} \right\rfloor = 25 \tag{5.2}$$

On the other hand, doubling the local batch size to $bs_{local} = 16$ increases the global batch size to 32, which mathematically halves the required synchronization frequency, dropping the number of events to 12 per epoch:

$$N_{step,bs=16} = \left\lfloor \frac{400}{16 \times 2} \right\rfloor = 12 \tag{5.3}$$

In both scenarios, the floor operator is introduced because Pytorch, by default, truncates the last incomplete batch.

While the frequency of synchronization events changes drastically, the physical dimension of the gradient tensor to be exchanged (S_{grad}) does not. The gradient size is intrinsically tied to the model architecture (number of parameters) and the numerical precision ($FP32$), completely independent of the batch size as defined in formula 3.12.

Therefore, the theoretical expectation for this comparative analysis is twofold: the network should transmit an identical payload of roughly 656 MB during each individual All-Reduce window in both scenarios, but the overall cumulative traffic generated across the entire epoch should be halved for $bs = 16$ due to the reduced number of communication steps.

Experimental Results and Traffic Analysis To empirically validate these theoretical expectations, the network traffic generated over a complete training epoch was captured and analyzed for both configurations. Table 5.6 presents the global network metrics, while Table 5.7 provides a statistical breakdown of the All-Reduce synchronization windows.

Global Epoch Metrics	**bs = 8** (FP32)	**bs = 16** (FP32)	**Ratio (16/8)**
Number of All-Reduce Events	25	12	0.48
Total bytes [MB]	16899.5	7945.0	0.47
Total duration [s]	582.5	325.1	0.56
$TH_{avg,overall}$ [Gbps]	0.232	0.196	0.84

Table 5.6: Overall network metrics for the complete training epoch comparing $bs = 8$ and $bs = 16$.

As visually confirmed by the cumulative traffic plot in Figure 5.15 and the metrics, scaling the local batch size effectively halves the global network traffic. The total data exchanged drops from approximately 16.9 GB to 7.9 GB (a measured ratio of 0.47), which perfectly correlates with the inverse proportional reduction in the required synchronization frequency (from 25 to 12 steps).

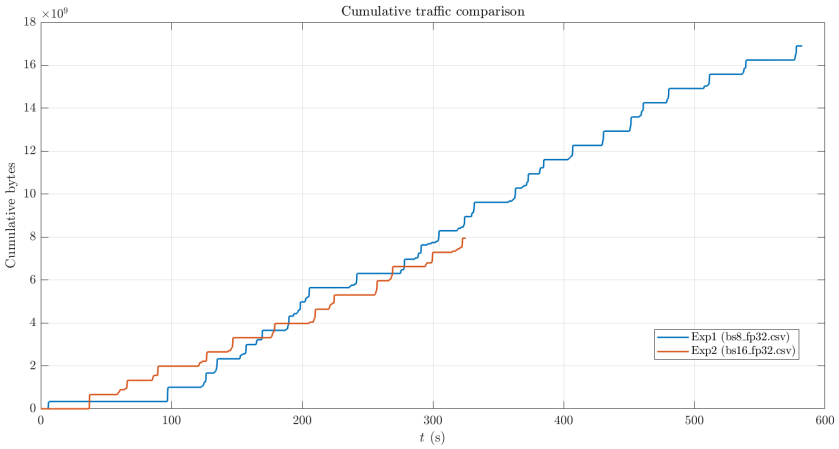


Figure 5.15: Cumulative network traffic comparison between $bs = 8$ and $bs = 16$ over a complete training epoch.

Furthermore, analyzing the behavior of the network, Figure 5.16 isolates the exact payload exchanged during each individual All-Reduce event. As mathematically predicted, the distribution is constant across both scenarios. The median payload transferred per event remains rigidly fixed at approximately 656.27 MB, regardless of the configured batch size. This analytically proves that the physical synchronization payload depends exclusively on the model's architectural dimension and numerical precision, effectively decoupling the network burst size from the computational workload volume processed in a single training step.

Per-Event Statistics	**bs = 8** (FP32)	**bs = 16** (FP32)	**Ratio (16/8)**
Number of AR events	25	12	0.480
Bytes/event (min) [MB]	656.248	656.258	1.000
Bytes/event (25%) [MB]	656.267	656.275	1.000
Bytes/event (median) [MB]	656.271	656.279	1.000
Bytes/event (75%) [MB]	656.279	656.284	1.000
Bytes/event (max) [MB]	656.296	656.290	1.000
Dur/event (min) [s]	0.626	0.682	1.090
Dur/event (25%) [s]	2.580	3.978	1.542
Dur/event (median) [s]	3.732	4.543	1.218
Dur/event (75%) [s]	5.490	6.103	1.112
Dur/event (max) [s]	8.961	10.511	1.173
TH_{peak} (min) [Mbps]	585.9	499.5	0.853
TH_{peak} (25%) [Mbps]	957.5	860.8	0.899
TH_{peak} (median) [Mbps]	1406.9	1155.7	0.821
TH_{peak} (75%) [Mbps]	2036.5	1320.9	0.649
TH_{peak} (max) [Mbps]	8392.9	7699.4	0.917

Table 5.7: Complete statistical breakdown of individual All-Reduce events comparing local batch sizes of 8 and 16 (FP32).

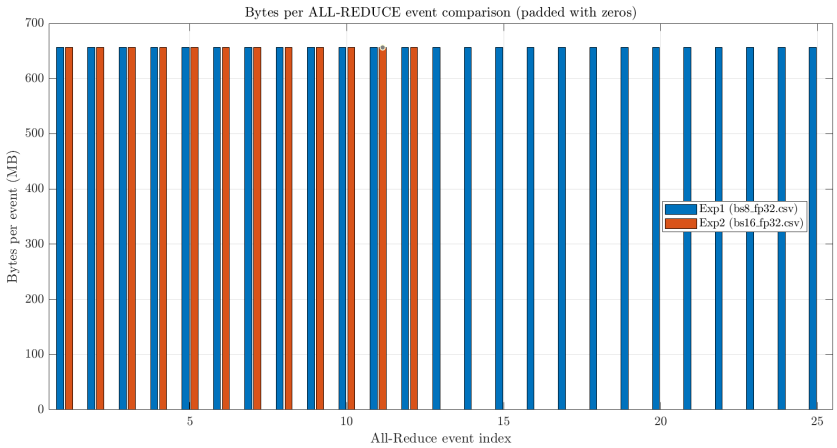


Figure 5.16: Bytes exchanged per individual All-Reduce event. The structural payload remains constant despite the batch size variation.

However, while the data volume is strictly deterministic, altering the batch size does impact the temporal dynamics of the transmission. As detailed in Table 5.7 and illustrated in Figure 5.17, the median peak throughput (TH_{peak}) experiences a noticeable

degradation, dropping from 1406.9 Mbps at $bs = 8$ to 1155.7 Mbps at $bs = 16$. Correspondingly, the median duration required to complete an All-Reduce synchronization slightly increases from 3.732 s to 4.543 s.

This behavioral shift is a direct consequence of the system moving deeper into a compute-bound regime. Processing 16 local samples per rank requires significantly more matrix multiplications during the backward pass. This extended computational phase alters the physiological overlap between gradient computation and the continuous DDP bucket transmission over the network. Consequently, the synchronization window is stretched, slightly lowering the instantaneous network efficiency per event.

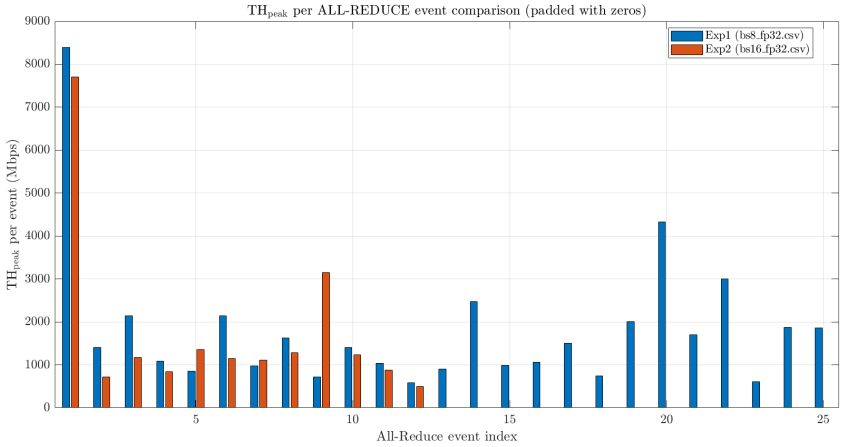


Figure 5.17: Peak throughput (TH_{peak}) recorded per event. The increased local computational load at $bs = 16$ stretches the communication window, lowering the instantaneous burst efficiency.

Impact on Wall-Clock Time and Communication Overhead To fully exploit the performance implications of the batch size scaling, it is analyzed the wall-clock execution time of the individual training steps. Table 5.8 reports the median and 75th percentile (p75) durations for the overall step (T_{step}), the isolated gradient synchronization window (T_{ar}), and the resulting Communication Overhead (OH_{ar}), defined as the percentage of the step time spent in network communication.

The data reveals a clear shift in the system's operational regime. When doubling the batch size from 8 to 16, the local computational workload per rank doubles, physiologically driving up the median total step time (T_{step}) from roughly 20.3 seconds to 29.4 seconds. Concurrently, the duration of the All-Reduce window (T_{ar}) experiences a minor increase (from 4.6 seconds to 5.3 seconds). As previously theorized, because the raw bytes exchanged remain strictly constant, this temporal stretch is exclusively caused by the altered overlap dynamics between the extended backward pass and the DDP bucketing mechanism, which slightly dilutes the network bursts over a longer timeframe.

Wall-Clock Metrics	**bs = 8** ($N = 25$)	**bs = 16** ($N = 12$)	**Ratio (16/8)**
T_{step} median [ms]	20296	29449	1.45
T_{step} p75 [ms]	29410	32470	1.10
T_{ar} median [ms]	4621	5327	1.15
T_{ar} p75 [ms]	5663	7696	1.36
OH_{ar} median [%]	27.8%	23.9%	0.86
OH_{ar} p75 [%]	35.5%	32.6%	0.92

Table 5.8: Wall-clock temporal metrics and communication overhead comparing local batch sizes of 8 and 16 (FP32).

However, the massive relative increase in computation time outpaces the minor communication penalty. Consequently, the median communication overhead (OH_{ar}) mathematically drops from 27.8% down to 23.9%. This proves that scaling the local batch size is a highly effective optimization strategy: by packing more computation into a single step, the system minimizes the frequency of synchronization penalties and pushes the distributed setup deeper into a compute-bound state, effectively hiding a larger fraction of the network latency, as suggested in Section 3.3.2.

Convergence and Algorithmic Trade-offs While scaling the global batch size optimizes the macroscopic network footprint and effectively reduces the communication overhead, it directly impacts the statistical convergence dynamics of the model. Profiling the training process reveals that the average training loss across the epoch increases from 2.4207 for the baseline $bs = 8$ scenario to 3.9135 for the $bs = 16$ configuration.

This behavior highlights a fundamental trade-off inherent to Distributed Data Parallel training. Because the overall dataset size is strictly fixed ($N_{samples} = 400$), doubling the global batch size physically halves the total number of gradient update steps per epoch (from 25 down to 12). With fewer algorithmic steps available to adjust its parameter weights via the optimizer, the model mathematically converges at a slower rate per epoch. Therefore, while increasing the batch size is highly beneficial from a purely networking and systemic perspective, minimizing the synchronization frequency, it has to guarantee the overall algorithmic convergence.

5.2.3 Precision Sensitivity Analysis: FP32 vs FP64

Having established the impact of computational scaling via batch size manipulation, the second dimension of the sensitivity analysis investigates the system's behavior under different numerical precision representations. This scenario directly compares the baseline 32-bit floating-point (FP32) configuration against a high-precision 64-bit floating-point (FP64) environment, strictly maintaining a constant local batch size of $bs = 8$ over the baseline $ws = 2$ topology.

Mathematical Expectations Unlike the batch size scaling, which dictates the frequency of the synchronization events, the numerical precision directly defines the structural dimension of the gradient tensor itself as mathematically formalized in 3.12. Since the model architecture remains identical, transitioning from FP32 to FP64 perfectly doubles the required bits per parameter.

Consequently, the theoretical expectation for this scenario is a strict doubling of the payload exchanged during each All-Reduce event ($B_{event,FP64} \approx 2 \times B_{event,FP32}$). Since the global batch size is unchanged, the total number of events remains fixed at exactly 25 per epoch, projecting an overall cumulative network footprint that is mathematically expected to be twice as large as the baseline.

Experimental Results and Traffic Analysis As in the previous case, Table 5.9 reports global metrics during training and Table 5.10 the statistics per event confirm what expected.

Global Epoch Metrics	**FP32**	**FP64**	**Ratio (64/32)**
Number of All-Reduce Events	25	25	1.00
Total bytes [MB]	16899.5	33635.0	1.99
Total duration [s]	582.5	934.6	1.60
$TH_{avg,overall}$ [Gbps]	0.232	0.288	1.24

Table 5.9: Overall network metrics for the complete training epoch comparing FP32 and FP64 precision ($bs = 8$).

Furthermore, the cumulative traffic plot in Figure 5.18 visually confirmed the expected results, since the overall traffic scales exactly as predicted, doubling from roughly 16.9 GB to 33.6 GB. In particular, analyzing the behavior of the network in Figure 5.19, the distribution of the bytes exchanged during each individual All-Reduce event is constant across the epoch. The median payload transferred per synchronization window shifts exactly from 656.27 MB to 1312.56 MB, yielding a perfectly rigid ratio of 2.00. This validates that the network burst size is intrinsically tied to the structural tensor size.

Per-Event Statistics	**FP32**	**FP64**	**Ratio (64/32)**
Number of AR events	25	25	1.000
Bytes/event (min) [MB]	656.248	1312.490	2.000
Bytes/event (25%) [MB]	656.267	1312.548	2.000
Bytes/event (median) [MB]	656.271	1312.560	2.000
Bytes/event (75%) [MB]	656.279	1312.567	2.000
Bytes/event (max) [MB]	656.296	1312.579	2.000
Dur/event (min) [s]	0.626	0.691	1.104
Dur/event (25%) [s]	2.580	6.507	2.522
Dur/event (median) [s]	3.732	9.041	2.423
Dur/event (75%) [s]	5.490	12.752	2.323
Dur/event (max) [s]	8.961	16.975	1.894
TH_{peak} (min) [Mbps]	585.9	618.6	1.056
TH_{peak} (25%) [Mbps]	957.5	823.5	0.860
TH_{peak} (median) [Mbps]	1406.9	1161.5	0.826
TH_{peak} (75%) [Mbps]	2036.5	1613.9	0.792
TH_{peak} (max) [Mbps]	8392.9	15205.8	1.812

Table 5.10: Complete statistical breakdown of individual All-Reduce events comparing FP32 and FP64 precision ($bs = 8$).

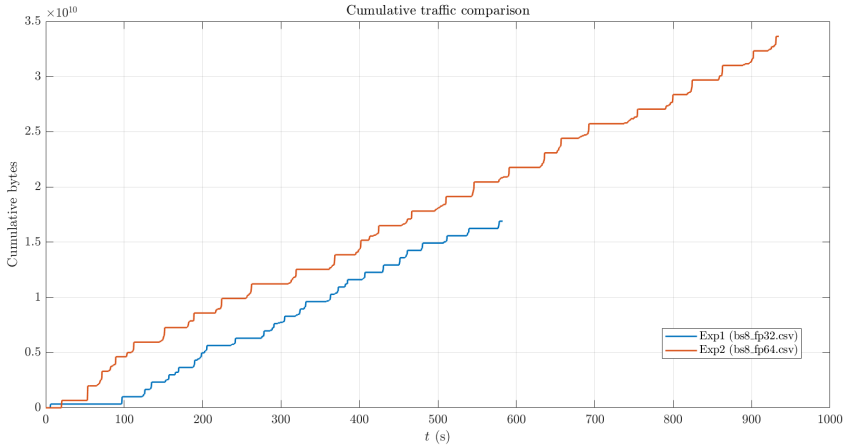


Figure 5.18: Cumulative network traffic comparison between FP32 and FP64 precision over a training epoch.

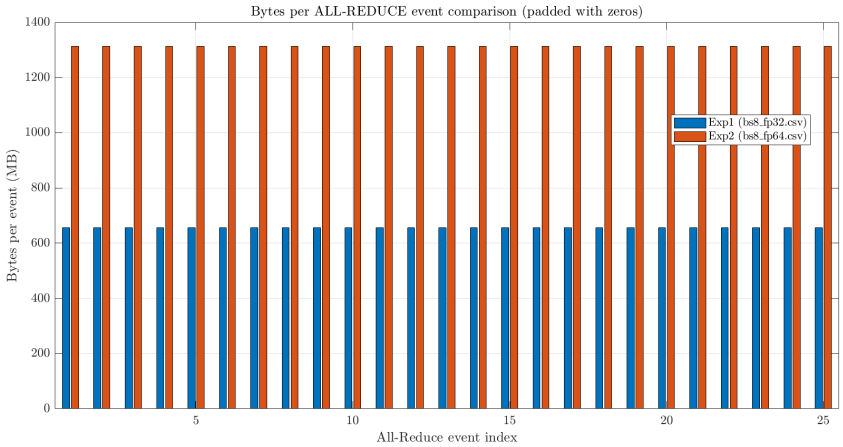


Figure 5.19: Bytes exchanged per individual All-Reduce event. The transition to FP64 perfectly doubles the structural network payload.

Impact on Wall-Clock Time and Communication Overhead While the volumetric scaling is strictly linear, the temporal scaling of the system exhibits a highly non-linear degradation, as reported in Table 5.11.

Wall-Clock Metrics	**FP32**	**FP64**	**Ratio (64/32)**
T_{step} median [ms]	20 296	37 347	1.84
T_{step} p75 [ms]	29 404	43 930	1.49
T_{ar} median [ms]	3 767	9 137	2.42
T_{ar} p75 [ms]	5 349	12 843	2.40
OH_{ar} median [%]	24.7%	27.3%	1.10
OH_{ar} p75 [%]	34.0%	35.2%	1.04

Table 5.11: Wall-clock temporal metrics and communication overhead comparing FP32 and FP64 ($bs = 8$).

The data reveals a severe asymmetry in the temporal domain. While the physical network payload increases by a factor of 2.00, the median duration of the All-Reduce event (T_{ar}) surges by a factor of 2.42 (from 3.76 s to 9.13 s). Consequently, as illustrated in Figure 5.20, the median peak throughput (TH_{peak}) drops from 1406.9 Mbps to 1161.5 Mbps. Simultaneously, the overall training step duration (T_{step}) also increases by a factor of 1.84 due to the heavier local computation. Because the communication time degrades more severely than the computation time, the median communication overhead OH_{ar} rises from 24.7% to 27.3%, pushing the system further away from the optimal compute-bound regime.

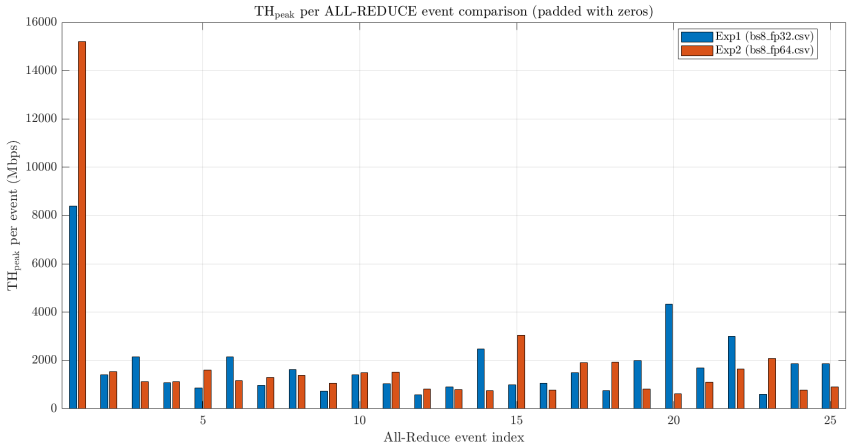


Figure 5.20: Peak throughput (TH_{peak}) recorded per event.

Hardware Microbenchmarking To justify this asymmetric temporal penalty, it is necessary to model the total duration of the All-Reduce event as the combination of data transmission, local mathematical reduction, memory access, and generic overhead ($T = T_{net} + T_{reduce} + T_{mem} + T_{overhead}$).

A dedicated C-based microbenchmark was developed to isolate the pure computational cost of the arithmetic operations at the local node level. Executing over 1.15×10^9 iterative multiplication and division by factor of 3 loops to simulate dense tensor reductions, the results demonstrated that the bare execution time organically increases from 9.95 s for 32-bit `float` variables to 11.06 s for 64-bit `double` variables (using unoptimized `-O0` compilation).

Applying this hardware-level evidence to the distributed training loop explains the 2.42× time inflation. The FP64 encoding not only forces the network to transmit double the bytes across the physical link (T_{net}) but concurrently demands more clock cycles from the CPU to perform the local gradient summations (T_{reduce}). Furthermore, moving 64-bit tensors generates heavier pressure on the internal memory bandwidth (T_{mem}). This combined computational and memory stress severely fragments the physiological overlap between the backward pass computation and the DDP bucketing mechanism. The result is an elongated synchronization window that exceeds the theoretical 2.0× expectation, lowering the instantaneous burst efficiency.

5.3 Experimental Analysis: World Size = 4

5.3.1 Scalability Setup

Topology Extension

To evaluate the scalability of the Distributed Data Parallel (DDP) paradigm and its structural impact on the underlying network infrastructure, the cluster architecture was expanded from the baseline $ws = 2$ configuration to a $ws = 4$ environment as visualized in

Figure 5.21. The experimental setup strictly maintained the baseline DistilGPT2 model, standard FP32 numerical precision, and a constant local batch size of $bs = 8$ across all participating computing nodes.

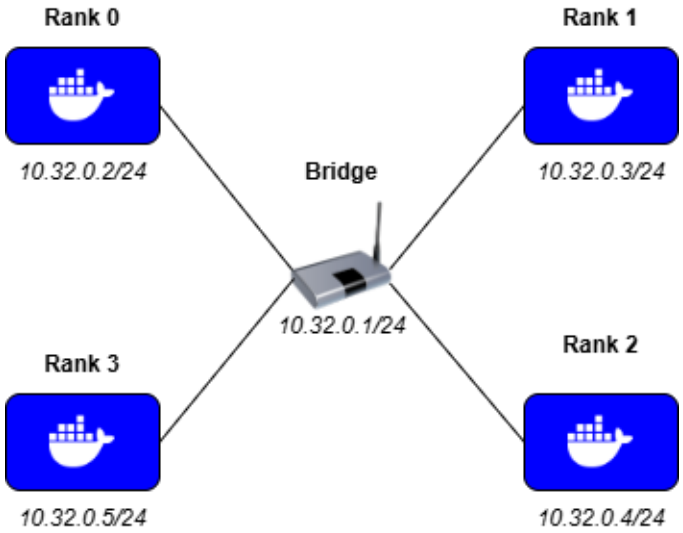


Figure 5.21: Physical Topology with $ws = 4$

Consistent with the baseline scenario, the network traffic exhibits a strictly bursty behavior. This temporal dynamic is clearly illustrated by the step-like progression of the cumulative traffic (Figure 5.22) and the isolated throughput spikes recorded during the synchronization windows (Figure 5.23).

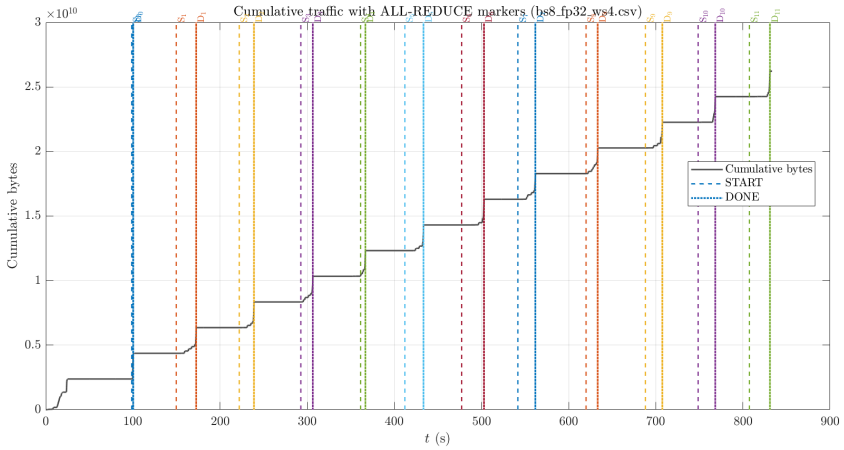


Figure 5.22: Cumulative network traffic over a single epoch ($ws = 4, bs = 8, FP32$).

By preserving a fixed local batch size while doubling the number of nodes, the global batch size processed doubles from 16 to 32 samples. Given the fixed dataset size of exactly

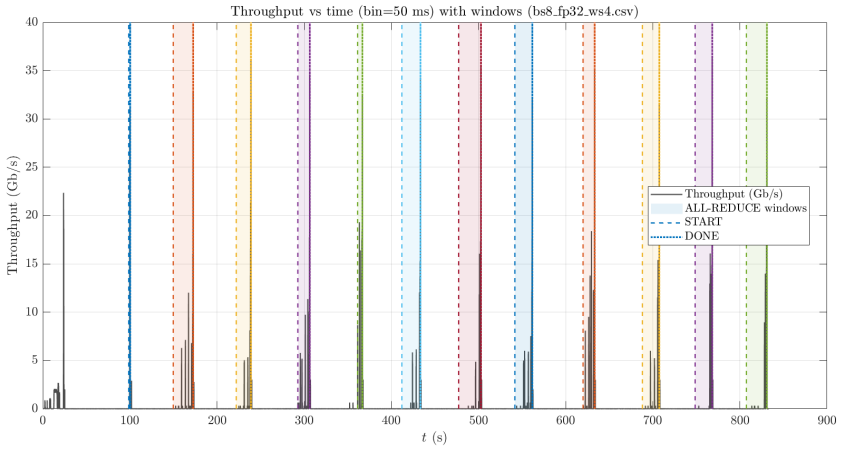


Figure 5.23: Instantaneous throughput per All-Reduce event.

400 samples, this architectural expansion mathematically halves the required number of training steps—and consequently, the frequency of the All-Reduce synchronization events, from 25 down to exactly 12 per epoch.

To ensure that the scalability analysis is not biased by local hardware bottlenecks, the system resource utilization was profiled across all four containers. As illustrated in Figure 5.24, the CPU utilization remains remarkably stable and heavily saturated, with each rank consistently drawing approximately 100% usage (effectively saturating one full core per process) throughout the entire training loop. Similarly, Figure 5.25 confirms a rigidly uniform memory allocated across the cluster, settling at roughly 2.8 GiB per node as in the baseline scenario with $ws = 2$.

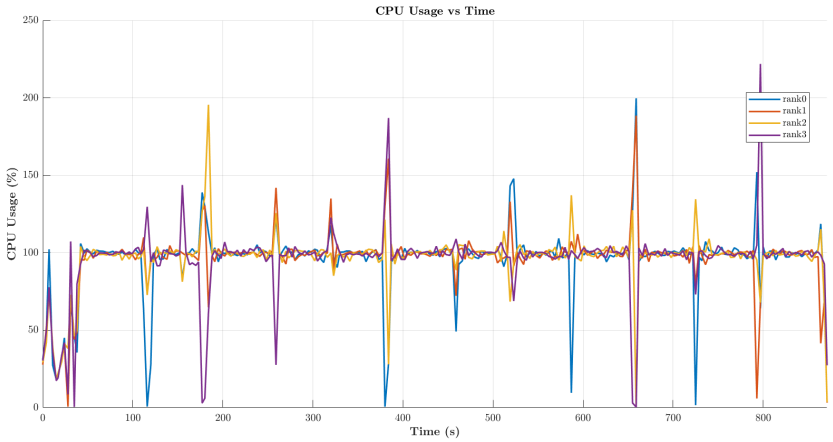


Figure 5.24: CPU utilization over time for each rank (WS=4).

Finally, the network I/O operations were monitored. Figure 5.26 tracks the cumulative transmitted (TX) and received (RX) traffic for each rank over time. The curves exhibit a

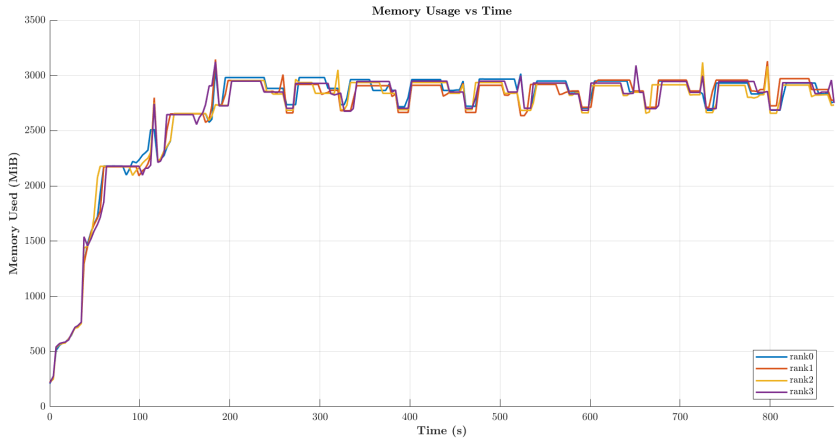


Figure 5.25: Memory consumption over time for each rank (WS=4).

steady, parallel, and step-like growth, which visually confirms the synchronization bursts. The asymmetries observed between the transmitted and received data across different ranks are generated by the initial broadcast.

The total cumulative traffic of approximately 6 GB transmitted by each rank corresponds exactly to the theoretical expectation of Formula 3.12. The data transmitted by each individual rank per step is equal to $\frac{2(N-1)S_{grad}}{N}$. For the expanded topology ($N = 4$), with a gradient size $S_{grad} \approx 328$ MB and an epoch consisting of $N_{steps} = 12$ steps, the total transmission per rank is calculated as:

$$B_{TX,event} = \frac{2(N-1)S_{grad}}{N} \times N_{steps} = \frac{2(4-1) \times 328}{4} \times 12 = 492 \times 12 = 5904 \text{ MB} \approx 6 \text{ GB} \tag{5.4}$$

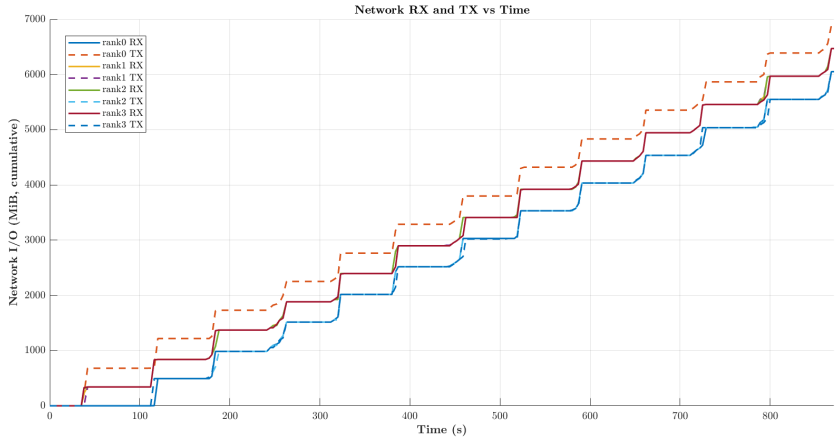


Figure 5.26: Comparison of cumulative transmitted and received traffic across ranks.

In particular, the 2-layer tree broadcast—introduced in Section 4.2.1—is evident in

this case. Specifically, the broadcast from rank 0 (master address 10.32.0.2/24) is sent directly to rank 1 (10.32.0.3/24) and rank 2 (10.32.0.4/24), while rank 1 subsequently propagates the message to rank 3 (10.32.0.5/24). This transmission pattern creates three different levels of cumulative traffic: the first relates to rank 0, which sends the initial broadcast packets plus the gradient transmissions during training; the second relates to rank 1, which sends one broadcast packet in addition to its gradient exchange; and the third corresponds to rank 2 and rank 3, which participate exclusively in the gradient synchronization. Figure 5.27 illustrates the traffic matrix of the information exchanged before the first All Reduce event.

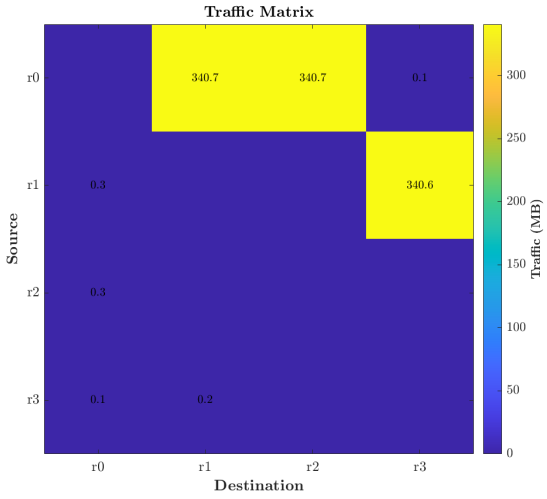


Figure 5.27: Traffic matrix of the initialization phase (rendezvous and broadcast) computed from packets captured before the first synchronization marker.

Unidirectional Ring All-Reduce Topology

To evaluate the gradient synchronization topology for the expanded $ws = 4$ cluster, the directional network traffic was analyzed. While a full-mesh physical connectivity is available between the containers, the distributed backend logically organizes the processes into a closed loop. The directional traffic matrix in Figure 5.29 illustrates this routing behavior, showing that the massive gradient exchange does not occur symmetrically across all node pairs. In Figure 5.28 the ring All-Reduce exchange is visualized.

This structural choice is confirmed by analyzing the cumulative traffic trends across all 8 possible directional flows, as shown in Figure 5.30 and Table 5.12. They highlight that four flows transmit the massive ≈ 6 GB gradient payload, while the remaining four flows carry minimal data, primarily consisting of TCP acknowledgments and the initial model broadcast. Specifically, the heavy traffic follows the sequential path `10.32.0.2` $\rightarrow$ `10.32.0.5` $\rightarrow$ `10.32.0.4` $\rightarrow$ `10.32.0.3` $\rightarrow$ `10.32.0.2` (corresponding to Rank 0 $\rightarrow$ Rank 3 $\rightarrow$ Rank 2 $\rightarrow$ Rank 1 $\rightarrow$ Rank 0). This specific pattern empirically proves the adoption of a Unidirectional Ring All-Reduce algorithm.

Focusing exclusively on the active ranks of the ring, Figure 5.31 reports the structural

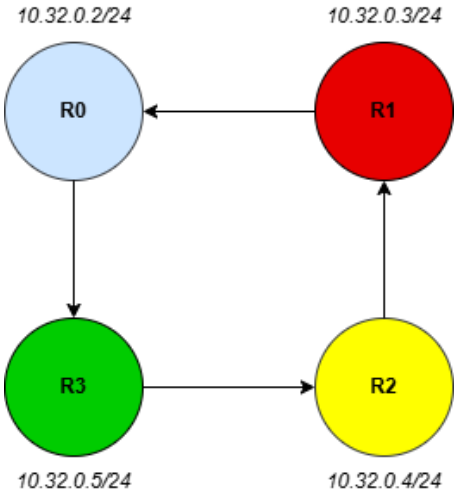


Figure 5.28: Ring All-Reduce exchange with $ws = 4$

payload transferred during each individual synchronization window. As mathematically expected for a unidirectional ring topology, the data transmitted by each active flow per event is approximately 492.1 MB. This value perfectly matches the theoretical formula 3.12.

Finally, Figure 5.32 presents the peak throughput (TH_{peak}) achieved by these four primary flows during the active transmission bursts, demonstrating how the active network bandwidth is exclusively utilized to forward the gradient chunks along the circular path.

Flow	**Total [MB]**	TH_{avg} **[Mbps]**	**Bytes/event [MB]**	TH_{peak} **[Mbps]**
Active Flows (Ring Path)				
02 → 05	5905.5	56.7	492.1	201.7
05 → 04	5904.8	58.2	492.1	201.7
04 → 03	5904.8	57.9	492.1	201.7
03 → 02	5905.7	56.7	492.1	201.7
Inactive/Control Flows				
02 → 03	413.5	4.0	0.3	0.1
05 → 02	3.1	< 0.1	0.2	0.1
03 → 04	3.0	< 0.1	0.3	0.1
04 → 05	2.7	< 0.1	0.2	0.1

Table 5.12: Directional breakdown of global metrics and per-event statistics for all 8 flows ($ws = 4$). Across the 12 total All-Reduce events, the median physical duration ($Dur/event$) observed is consistently 19.519 s. The flow 02 → 03 records a higher total volume due to the initial model broadcast.

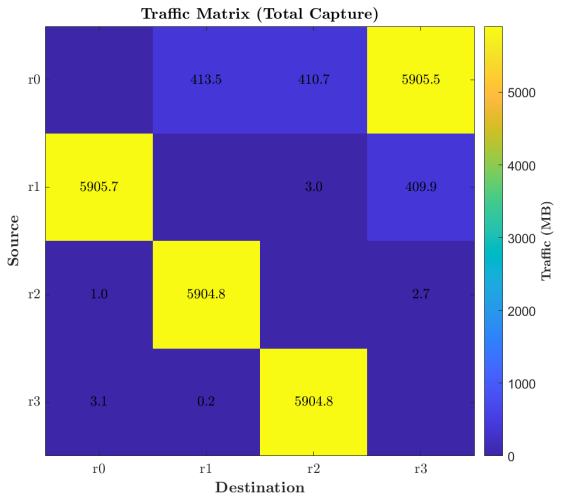


Figure 5.29: Directional traffic matrix illustrating the logical ring topology during the $ws = 4$ training session.

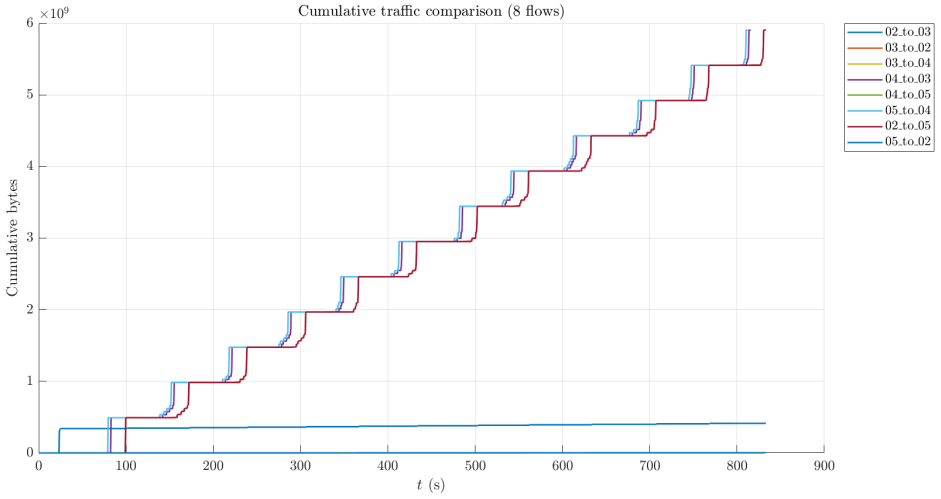


Figure 5.30: Cumulative network traffic comparison across the 8 directional flows.

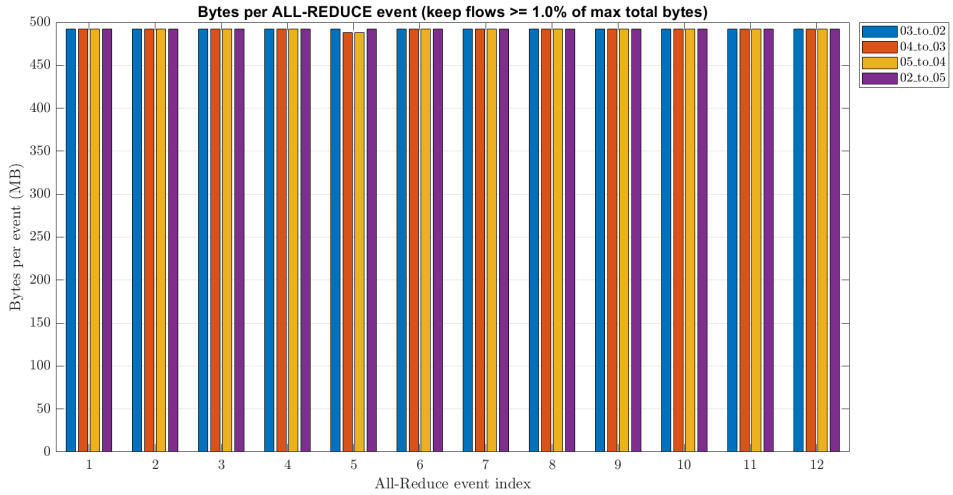


Figure 5.31: Bytes exchanged per All-Reduce event for the active flows.

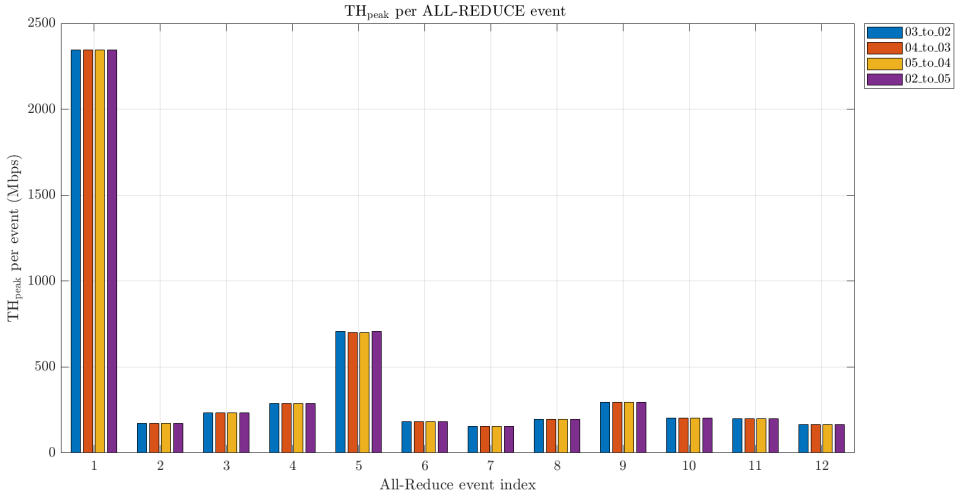


Figure 5.32: Peak throughput (TH_{peak}) per event for the active directional flows along the ring.

5.3.2 Cumulative Traffic Comparison (WS=2 vs WS=4)

Mathematical Expectations

When expanding the cluster size from $ws = 2$ to $ws = 4$ while maintaining a fixed local batch size ($bs = 8$) and precision (FP32), two distinct algorithmic scaling behaviors occur simultaneously: a reduction of the steps during the overall training and an increase in the traffic exchange during a gradient synchronization.

First, by doubling the number of participating nodes, the global batch size processed

by the cluster at each step effectively doubles (from 16 to 32 samples). Because the overall dataset size remains static, processing more samples per step mathematically reduces the total number of required training iterations per epoch. Based on the theoretical expectations, the number of steps (and consequently the number of All-Reduce events) drops from 25 to 12.

Second, the total volume of data exchanged per node during each isolated synchronization window scales proportionally to the number of participants N. According to the theoretical model of the Ring All-Reduce algorithm, the payload per event follows the relation 3.12. When transitioning from a 2-node to a 4-node topology, this formulation mathematically predicts a precise 3× expansion in the per-event payload:

$$\frac{B_{ws4}}{B_{ws2}} = \frac{2(4-1)}{2(2-1)} = 3 \tag{5.5}$$

Consequently, the macroscopic cumulative traffic across the entire epoch is expected to be the result of these two contrasting factors: a payload per event that triples, partially mitigated by the number of events that roughly halves. The expected overall volume growth is therefore a factor of approximately 1.5×.

Experiments Results and Traffic Analysis

To empirically validate the theoretical expectations, the network traffic generated over a complete training epoch was captured and analyzed. Table 5.13 presents the global network metrics, while Table 5.14 provides a detailed statistical breakdown of the individual All-Reduce synchronization windows.

Global Epoch Metrics	**WS=2**	**WS=4**	**Ratio (4/2)**
Number of All-Reduce Events	25	12	0.48
Total bytes [MB]	16899.5	26222.0	1.55
Total duration [s]	582.5	833.4	1.43
$TH_{avg,overall}$ [Gbps]	0.232	0.252	1.08

Table 5.13: Overall network metrics for the complete training epoch comparing $ws = 2$ and $ws = 4$.

Analyzing the total bytes exchanged, the total cumulative traffic effectively increases from 16.9 GB in the baseline scenario to 26.2 GB in the $ws = 4$ configuration. This represents an overall volume growth of 1.55×, accurately reflecting the mathematical model. This step-like progression is clearly illustrated in Figure 5.33.

Furthermore, Figure 5.34 validates the structural properties of the Ring All-Reduce. The median data transferred during a single burst jumps perfectly by a factor of 3, from 656.3 MB to 1969.3 MB, proving that the physical synchronization payload scales precisely according to the algorithmic properties of the distributed topology.

While the traffic scaling is strictly deterministic, the temporal domain experiences a significant performance degradation. As observed in Table 5.14, the median physical

Per-Event Statistics	**WS=2**	**WS=4**	**Ratio (4/2)**
Bytes/event (min) [MB]	656.248	1961.115	2.99
Bytes/event (median) [MB]	656.271	1969.272	3.00
Bytes/event (max) [MB]	656.296	1969.339	3.00
Dur/event (median) [s]	3.732	19.519	5.23
Dur/event (max) [s]	8.961	25.571	2.85
TH_{peak} (median) [Mbps]	1406.9	807.2	0.57
TH_{peak} (max) [Mbps]	8392.9	9384.5	1.12

Table 5.14: Statistical breakdown of the structural payload and instantaneous throughput exchanged per event between $ws = 2$ and $ws = 4$.

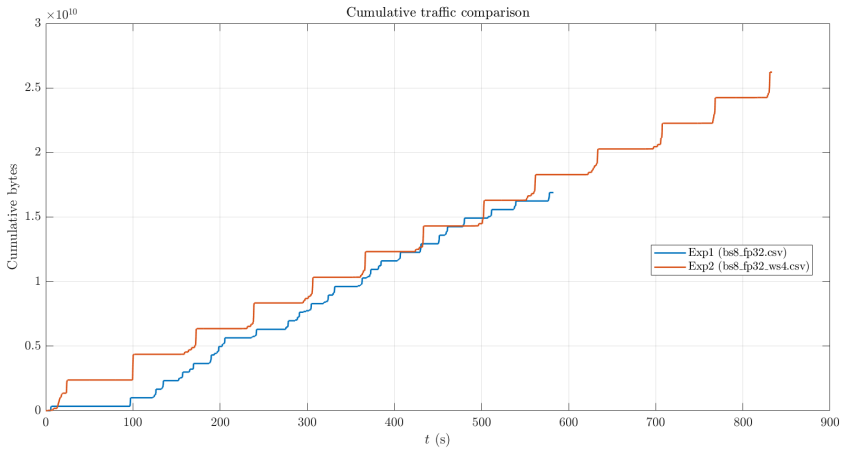


Figure 5.33: Cumulative network traffic comparison between $ws = 2$ and $ws = 4$ topologies. The expanded cluster generates $\approx 1.55\times$ more total traffic despite executing fewer synchronization steps.

duration of the network burst ($Dur/event$) escalates from 3.73 seconds to 19.52 seconds, leading to this massive 5.23× temporal degradation. Consequently, as shown in Figure 5.35, the median peak throughput drops significantly from 1406.9 Mbps to 807.2 Mbps.

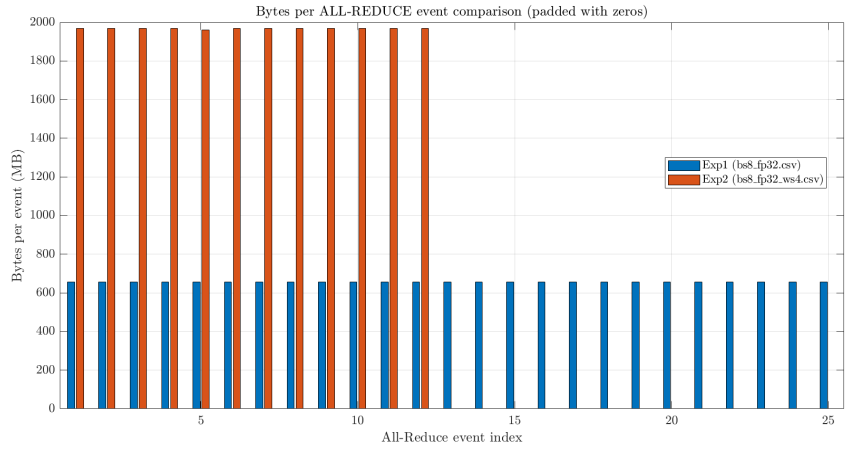


Figure 5.34: Bytes exchanged per individual All-Reduce event. The payload increases by a factor of 3, adhering exactly to the $2(N-1)S_{grad}$ scaling rule.

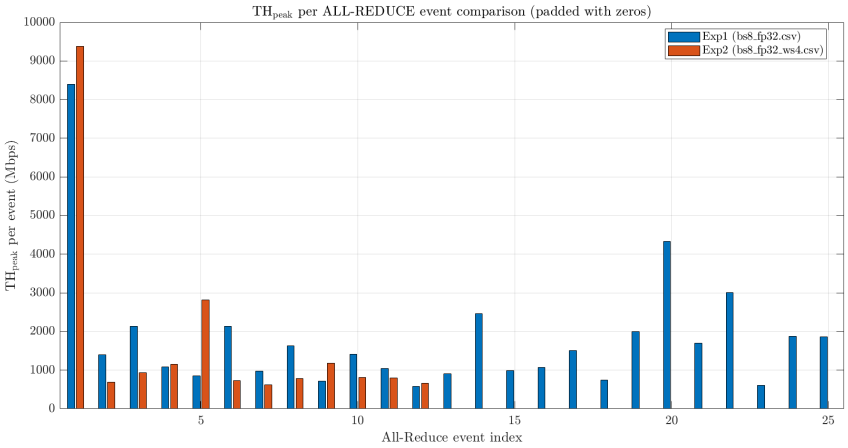


Figure 5.35: Peak throughput (TH_{peak}) per event comparison. The $ws = 4$ topology experiences a drop in median throughput due to the increased serialization of the logical ring.

Wall-Clock Metrics	**WS=2**	**WS=4**	**Ratio (4/2)**
T_{step} median [ms]	20296	66989	3.30
T_{step} p75 [ms]	29410	71654	2.44
T_{ar} median [ms]	4621	21010	4.55
T_{ar} p75 [ms]	5663	23175	4.09
OH_{ar} median [%]	27.8%	33.0%	1.19
OH_{ar} p75 [%]	35.5%	36.4%	1.03

Table 5.15: Wall-clock temporal metrics and communication overhead comparing $ws = 2$ and $ws = 4$ ($bs = 8$, FP32).

This network inefficiency severely impacts the application layer. Wall-clock metrics detailed in Table 5.15 show that the median total step time (T_{step}) triples from 20.3 seconds to 67.0 seconds. More importantly, the median software-level All-Reduce duration (T_{ar}) increase by a factor of 4.55 (from 4.6 s up to 21.0 s), driven by the strict serialization of the chunks propagating through the extended ring. Ultimately, while the system remains primarily compute-dominated, the communication overhead (OH_{ar}) rises from 27.8% to 33.0%, highlighting that network constraints become progressively more dominant as the cluster scales.

5.3.3 Performance Degradation

To rigorously evaluate the network's resilience from possible bottlenecks during training, network degradation scenarios were emulated on the $ws = 4$ topology ($bs = 8$, FP32). Using the Linux `tc` (Traffic Control) utility, artificial constraints were injected exclusively on a single active edge of the logical ring. Specifically, the traffic leaving Rank 3 (`10.32.0.5/24`) and destined for Rank 2 (`10.32.0.4/24`) was targeted.

In a strictly synchronous Ring All-Reduce algorithm, the total synchronization time is dictated by the slowest link in the loop. Consequently, degrading a single flow forces a global temporal increase, causing the entire cluster to stall.

Bandwidth Limitation Impact

In the first phase, artificial bandwidth of 200 Mbps, 100 Mbps, and 50 Mbps were enforced on the target link, comparing the results against the unconstrained baseline.

As mathematically expected, Table 5.17 and Figure 5.38 confirm that the network exchanged payload remains rigidly stable at roughly 1968 MB per event, as the model architecture dictates the gradient size. The impact of the bottleneck manifests entirely in the temporal domain. As the bandwidth set becomes smaller, the physical duration of the All-Reduce burst ($Dur/event$) drastically increase from 19.5 s up to 87.7 s.

This temporal stretch degrades the overall network efficiency. As illustrated by the bar charts in Figure 5.37 and in Table 5.16, the global average throughput ($TH_{avg,overall}$) drops progressively from 0.252 Gbps to 0.114 Gbps. Similarly, the instantaneous median peak throughput (TH_{peak}) drops from 807.2 Mbps down to 179.0 Mbps.

Consequently, at the application layer, the entire DDP pipeline is limited. As detailed in Table 5.18, the median All-Reduce time (T_{ar}) increase by a factor of over 4.3× (from 21.0 s to 91.7 s). Because the computation phase remains constant, this directly forces the Communication Overhead (OH_{ar}) to rise from a baseline of 33.0% up to an unsustainable 64.0%, effectively pushing the distributed setup from a compute-bound into a severely communication-bound regime.

Global Epoch Metrics	**BASE**	**BW 200**	**BW 100**	**BW 50**
Number of All-Reduce Events	12	12	12	12
Total bytes [MB]	26222.0	26250.8	26304.7	24877.2
Total duration [s]	833.4	1072.4	1283.8	1745.0
$TH_{avg,overall}$ [Gbps]	0.252	0.196	0.164	0.114

Table 5.16: Overall network metrics comparing the baseline against localized bandwidth caps.

Per-Event Statistics	**BASE**	**BW 200**	**BW 100**	**BW 50**
Bytes/event (median) [MB]	1969.3	1964.2	1962.0	1964.1
Bytes/event (max) [MB]	1969.3	1966.6	1966.3	1964.5
Dur/event (median) [s]	19.52	28.89	42.51	87.75
Dur/event (max) [s]	25.57	42.86	61.39	92.05
TH_{peak} (median) [Mbps]	807.2	544.1	368.9	179.0
TH_{peak} (max) [Mbps]	9384.5	691.1	385.7	193.5

Table 5.17: Statistical breakdown of the structural payload and throughput per event under bandwidth restrictions.

Wall-Clock Metrics	**BASE**	**BW 200**	**BW 100**	**BW 50**
T_{step} median [ms]	66 989	86 372	102 680	145 580
T_{step} p75 [ms]	71 654	90 407	106 500	147 460
T_{ar} median [ms]	21 010	38 704	54 904	91 736
T_{ar} p75 [ms]	23 175	42 623	58 295	99 617
OH_{ar} median [%]	33.0	44.6	49.5	64.0
OH_{ar} p75 [%]	36.4	49.3	60.0	72.1

Table 5.18: Wall-clock temporal metrics and communication overhead resulting from bandwidth degradation.

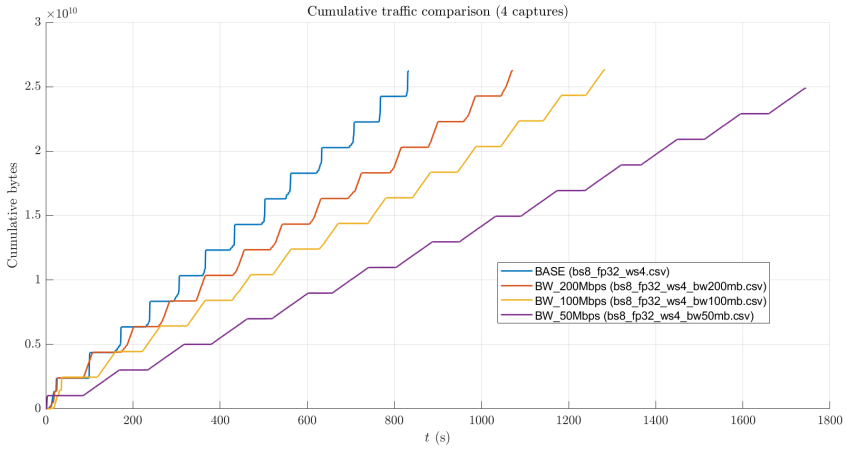


Figure 5.36: Cumulative traffic progression. The step increments become progressively more diluted as the bottleneck becomes stricter.

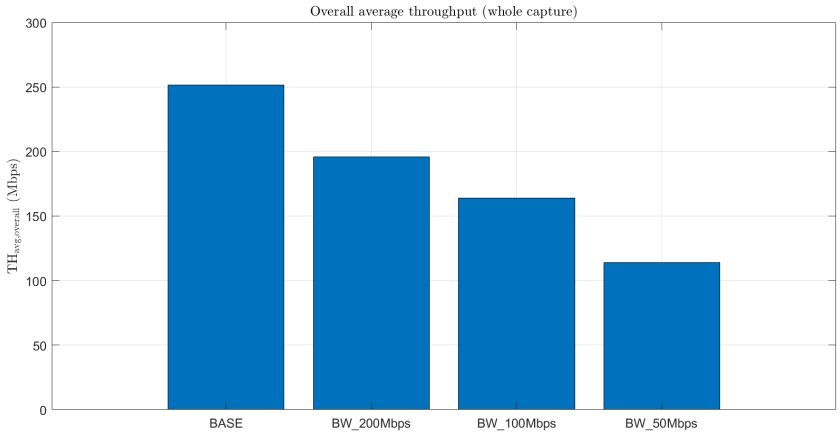


Figure 5.37: Overall average throughput ($TH_{avg,overall}$) drop resulting from the artificial bandwidth caps.

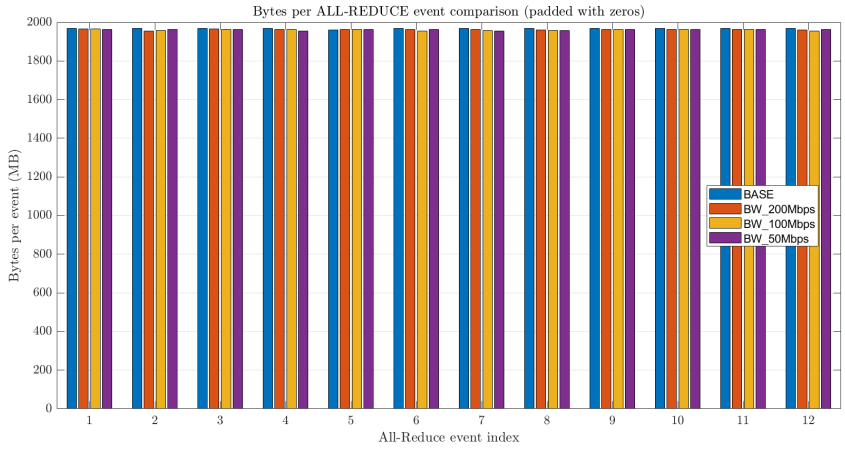


Figure 5.38: Bytes exchanged per event. The structural payload remains invariant despite the network constraints.

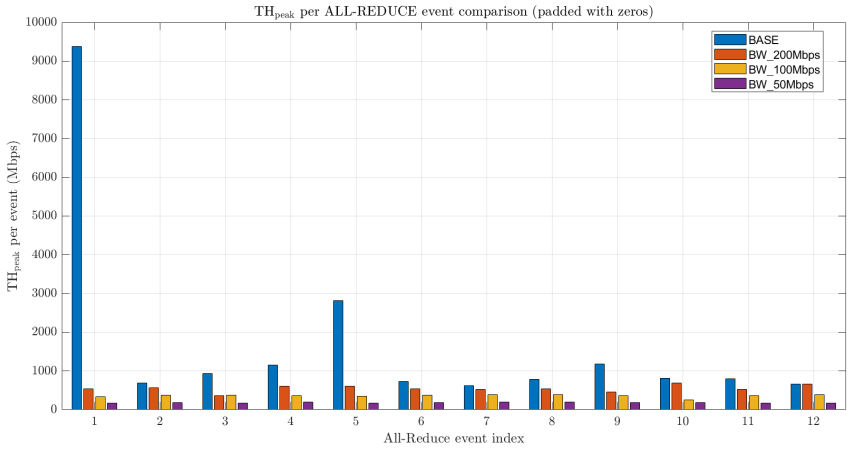


Figure 5.39: Peak throughput (TH_{peak}) per event. The artificial bottleneck forces a severe drop in the instantaneous burst efficiency.

Latency Injection Impact

In the second phase, constant artificial delays (100 ms, 200 ms, and 300 ms) were injected at the packet level into the target link using `netem`. Unlike bandwidth case, which directly restricts traffic over time, latency injection disrupts the synchronization by forcing the TCP windows to wait longer for acknowledgments, delaying the propagation of the gradient chunks along the ring.

The results follow the same theoretical pattern observed in the bandwidth experiments: the physical payload size remains invariant ($\approx$ 1969 MB), while the temporal metrics dramatically inflate. As detailed in Table 5.20, adding a 300 ms delay per packet

pushes the median physical network duration ($Dur/event$) from 19.5 s to 64.6 s, effectively dropping the average global throughput to 0.135 Gbps (Table 5.19).

At the application layer (Table 5.21), this artificial latency cascades through the pipelined updates, elevating the median All-Reduce time (T_{ar}) to 76 s. Consequently, the Communication Overhead (OH_{ar}) reaches 60.7%, proving that high-latency segments in a distributed AI fabric break the computational overlap, stalling the entire cluster even if the raw bandwidth capacity is theoretically sufficient.

Global Epoch Metrics	**BASE**	**LAT 100**	**LAT 200**	**LAT 300**
Number of All-Reduce Events	12	12	12	12
Total bytes [MB]	26222.0	24880.9	26174.1	26319.9
Total duration [s]	833.4	1266.2	1336.6	1554.9
$TH_{avg,overall}$ [Gbps]	0.252	0.157	0.157	0.135

Table 5.19: Overall network metrics comparing the baseline against localized latency injection.

Per-Event Statistics	**BASE**	**LAT 100**	**LAT 200**	**LAT 300**
Bytes/event (median) [MB]	1969.3	1963.0	1964.5	1966.1
Bytes/event (max) [MB]	1969.3	1969.7	1967.7	1973.6
Dur/event (median) [s]	19.52	32.43	49.54	64.60
Dur/event (max) [s]	25.57	167.72	56.77	91.46
TH_{peak} (median) [Mbps]	807.2	484.2	317.3	243.4
TH_{peak} (max) [Mbps]	9384.5	710.0	396.1	271.5

Table 5.20: Statistical breakdown of the structural payload and throughput per event under latency injection.

Wall-Clock Metrics	**BASE**	**LAT 100**	**LAT 200**	**LAT 300**
T_{step} median [ms]	66 989	92 753	108 090	122 860
T_{step} p75 [ms]	71 654	96 551	111 490	124 510
T_{ar} median [ms]	21 010	41 879	53 676	76 027
T_{ar} p75 [ms]	23 175	48 939	55 544	83 029
OH_{ar} median [%]	33.0	46.2	48.9	60.7
OH_{ar} p75 [%]	36.4	59.2	56.4	66.9

Table 5.21: Wall-clock temporal metrics and communication overhead resulting from latency injection.

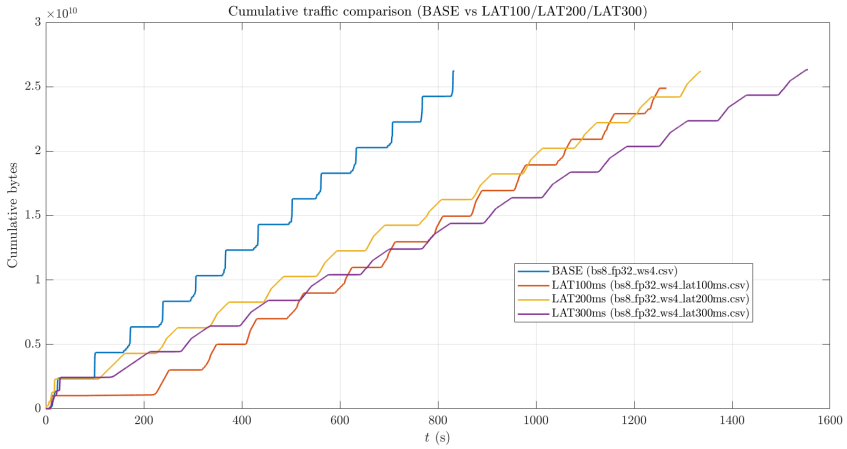


Figure 5.40: Cumulative traffic progression. The step increments become progressively more diluted as the artificial latency increases.

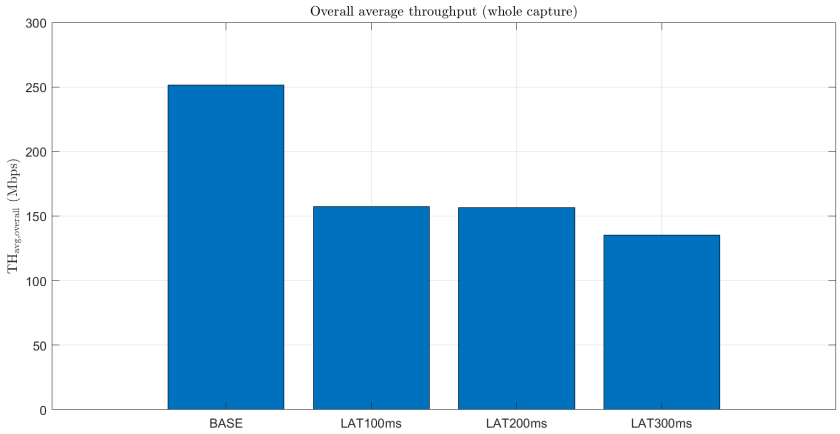


Figure 5.41: Overall average throughput ($TH_{avg,overall}$) drop resulting from the artificial latency injection.

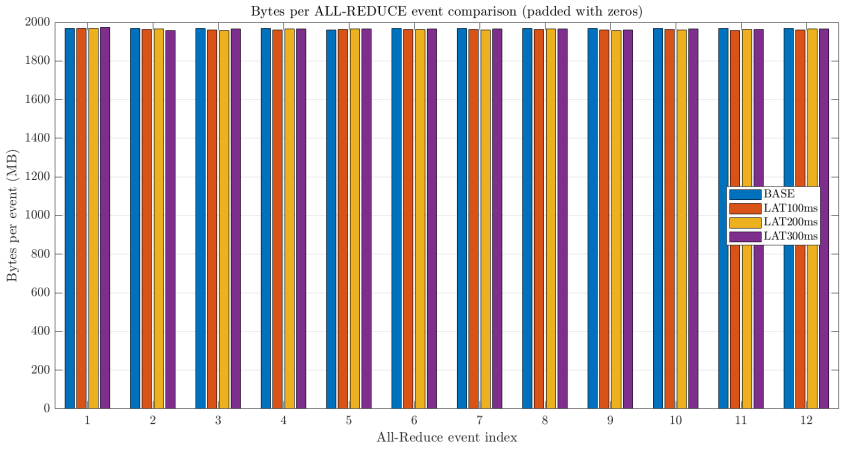


Figure 5.42: Bytes exchanged per event. The structural payload remains invariant despite the network constraints.

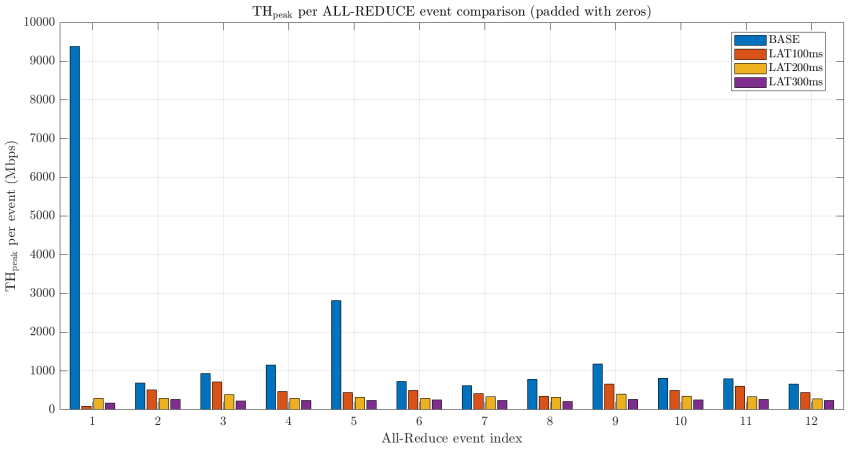


Figure 5.43: Peak throughput (TH_{peak}) per event. The artificial delay forces a severe drop in the instantaneous burst efficiency.

The observed performance degradation under artificial latency closely mirrors the impact of direct bandwidth restriction. This equivalence stems from the limitations of the TCP maximum congestion window, which cannot scale sufficiently to sustain the high throughput required for training when the Round-Trip Time (RTT) increases. Consequently, the injected latency acts effectively as a severe bandwidth bottleneck.

Chapter 6

Conclusion

The necessity to scale Large Language Models beyond the limits of single, monolithic data centers, overcoming power availability and hardware constraints, motivated the foundation of this thesis.

This work introduced and analyzed the distributed AI training problem, focusing on the transition from centralized AI clusters to geographically decentralized AI fabrics.

Chapter 2 introduces the main characteristics of Large Language Models and shows the most relevant solutions that have been proposed in the literature to move the training process from a single node towards a distributed approach. The chapter then describes in detail the computational dynamics of the training and inference phases. Finally, we review the current state-of-the-art networking infrastructures for distributed AI and inherent challenges of this field.

In chapter 3, we explain the network requirements and properties of distributed training, which is the main focus of this work. We then delineate our theoretical framework based on the scenario discussed in Epoch AI article. We brought our main contribution to the model by mathematically decomposing the distributed training time into distinct computation and synchronization phases.

Chapter 4 describes how the system model was implemented in an experimental environment using reproducible components. We started from the PyTorch DistributedDataParallel (DDP) library with the Gloo backend and we set up the environment to isolate the Ring All-Reduce traffic. We encapsulated the AI workers inside Docker containers to control the network connectivity and the resources usage. We eventually connected the workers to a virtual Linux bridge to obtain a network topology easier to analyze.

In chapter 5, we describe the experimental setup and we show and comment the results. We performed different kinds of experiments: we firstly evaluated the network footprint by varying critical hyperparameters, namely the batch size, the floating-point precision, and the world size. Subsequently, we evaluated the framework's resilience by introducing targeted network degradations, observing the system's breaking points under severe bandwidth and latency bottlenecks.

6.1 Key Findings and Lessons Learned

Predictability of the Distributed AI Traffic Matrix

One of the most significant findings of this work is the strict predictability of the network traffic matrix in decentralized AI training. The experimental results confirmed that the synchronization payload is entirely deterministic: it depends exclusively on the model's architecture, numerical precision, and cluster size, remaining completely decoupled from the local batch size. Furthermore, the flow distribution proved to follow a rigid ring topology. From a network engineering perspective, this strict determinism is crucial, as it allows architects to accurately forecast the communication overhead and proactively provision WAN capacity without relying on empirical approximations.

Effects of Network Latency

Furthermore, the degradation analysis revealed non-trivial effects of network latency on the training loop. While bandwidth bottlenecks are an intuitive limitation, our results demonstrated that high-latency segments are equally destructive, even when raw bandwidth is high. Increased latency disrupts the pipelined synchronization of the DDP framework, completely breaking the physiological overlap between computation and gradient communication. Consequently, the communication overhead spikes to unsustainable levels (over 60%), causing the computational units to stall and pushing the distributed architecture out of an optimal compute-bound regime into a severe communication-bound state.

Non-Linear Temporal Penalty of High Precision

The sensitivity analysis regarding numerical precision (FP32 vs. FP64) highlighted an asymmetry in system performance. While the physical network payload scales linearly—exactly doubling when transitioning to 64-bit tensors—the temporal degradation is highly non-linear. The All-Reduce duration increased by a factor of 2.42x. This demonstrates that transmitting heavier payloads stresses not only the network link but also the local memory bandwidth and CPU computational capacity. This combined hardware strain severely fragments the physiological overlap between the backward pass and the DDP bucketing mechanism, elongating the synchronization window beyond theoretical network expectations.

Scalability Bottlenecks in Ring Topologies

Finally, the cluster expansion from two to four nodes empirically validated the structural limits of the Ring All-Reduce algorithm over standard TCP/IP stacks. While the payload size per event perfectly matched the theoretical $2(N-1)/N$ scaling rule (tripling the exchanged bytes), the temporal penalty was disproportionately high, with burst durations increasing by over 5x. This serialization inherently increase the communication overhead (OH_{ar}), pushing the distributed setup further away from the compute-bound regime. This finding proves that as the decentralized cluster scales geographically, the network serialization delay quickly becomes the dominant bottleneck, highlighting the absolute necessity for specialized transport optimizations like Kernel Bypass or RDMA in larger deployments.

6.2 Future directions

Although the experimental framework proposed in this thesis successfully characterizes the impact of network constraints on the Distributed Data Parallel (DDP) paradigm, it represents a baseline architecture that can be significantly expanded to mirror modern, industrial-scale decentralized AI fabrics.

From an infrastructure and hardware perspective, the natural evolution of this work involves migrating from CPU-based containerization with the Gloo backend to a fully hardware-accelerated environment utilizing GPUs and the NVIDIA Collective Communications Library (NCCL). Modern large-scale training relies on Remote Direct Memory Access (RDMA) over Converged Ethernet (RoCE), which enables Kernel Bypass and GPUDirect technologies. Evaluating the gradient synchronization in such an environment would remove the Operating System TCP/IP stack overhead, allowing for a precise characterization of latencies and network congestion.

Secondly, scaling the computational workload to more realistic, state-of-the-art Large Language Models (involving billions of parameters) necessitates the adoption of more complex parallelization strategies. While this study focused purely on Data Parallelism, future research should integrate Tensor and Pipeline Parallelism. This would enable the observation of how diverse parallelization strategies generate heterogeneous traffic patterns across the network, and how different solutions best adapt to specific scenarios.

To further mitigate the communication bottlenecks observed during the cluster expansion, different topological solutions for gradient synchronization could be investigated.

Finally, the experimental environment could be scaled in terms of the number of participating processes and the injection of potential fault events. By utilizing advanced models and more heterogeneous datasets, future research could deeply analyze the systemic performance and, consequently, evaluate effective optimization solutions during both training and inference.

Bibliography

[1] Yangtao Deng, Xiang Shi, Zhuo Jiang, Xingjian Zhang, Lei Zhang, Zhang Zhang, Bo Li, Zuquan Song, Hang Zhu, Gaohong Liu, Fuliang Li, Shuguang Wang, Haibin Lin, Jianxi Ye, and Minlan Yu. Minder: Faulty machine detection for large-scale distributed model training. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2025.

[2] Haotian Dong, Jingyan Jiang, Rongwei Lu, Jiajun Luo, Jiajun Song, Bowen Li, Ying Shen, and Zhi Wang. Beyond a single AI cluster: A survey of decentralized LLM training. *arXiv*, 2025.

[3] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, Shuqiang Zhang, Mikel Jimenez Fernandez, Shashidhar Gandham, and Hongyi Zeng. Rdma over ethernet for distributed ai training at meta scale. In *Proceedings of the ACM SIGCOMM 2024 Conference*. ACM, 2024.

[4] Hugging Face. Byte-pair encoding tokenization. https://huggingface.co/learn/llm-course/en/chapter6/5. Hugging Face LLM Course, Chapter 6. Accessed: 2026-03-01.

[5] Hugging Face. DistilGPT2 model card. https://huggingface.co/distilbert/distilgpt2. Hugging Face Model Hub. Accessed: 2026-03-02.

[6] Jaykumaran. Distributed parallel training: Pytorch multi-gpu setup in kaggle t4x2. https://learnopencv.com/distributed-parallel-training-pytorch-multi-gpu-setup/. Accessed: 2026-02-17.

[7] Aruna Kolluru. Model parallelism. https://medium.com/@aruna.kolluru/model-parallelism-390d32145a5a. Medium Blog. Accessed: 2026-02-17.

[8] Ferdi Kossmann, Bruce Fontaine, Daya Khudia, Michael Cafarella, and Samuel Madden. Is the GPU half-empty or half-full? practical scheduling techniques for LLMs. *arXiv*, 2025.

[9] Vipul Koti. Attention for your transformers. https://medium.com/@vipul.koti333/attention-for-your-transformers-632cc23ca973. Medium Blog. Accessed: 2026-03-01.

[10] Vipul Koti. From theory to code: Step-by-step implementation and code breakdown of GPT-2 model. https://medium.com/@vipul.koti333/from-theory-to-code-step-by-step-implementation-and-code-breakdown-of-gpt-2-model-7bde8d5cecda. Medium Blog. Accessed: 2026-02-17.

[11] Vipul Koti. Word embeddings (nlp view). https://medium.com/@vipul.koti333/word-embeddings-nlp-view-a7c80bc7353c. Medium Blog. Accessed: 2026-03-01.

[12] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1608.05859*, 2016.

[13] Michael Mitzenmacher and Rana Shahout. Queueing, predictions, and large language models: Challenges and open problems. *Stochastic Systems*, 15(3):195–219, 2025.

[14] NVIDIA. AI fabric resiliency and why network convergence matters. https://developer.nvidia.com/blog/ai-fabric-resiliency-and-why-network-convergence-matters/. NVIDIA Developer Blog. Accessed: 2026-02-17.

[15] NVIDIA. AI factory. https://www.nvidia.com/en-us/glossary/ai-factory/. NVIDIA Glossary. Accessed: 2026-02-17.

[16] NVIDIA. How to connect distributed data centers into large AI factories with scale-across networking. https://developer.nvidia.com/blog/how-to-connect-distributed-data-centers-into-large-ai-factories-with-scale-across-networking. Accessed: 2026-02-17.

[17] NVIDIA. Introducing NVIDIA dynamo – a low latency distributed inference framework for scaling reasoning AI models. https://developer.nvidia.com/blog/introducing-nvidia-dynamo-a-low-latency-distributed-inference-framework-for-scaling-reasoning-ai-models/. Accessed: 2026-02-17.

[18] NVIDIA. Maximizing low-latency networking performance for financial services with NVIDIA rivermax and neio fastsocket. https://developer.nvidia.com/blog/maximizing-low-latency-networking-performance-for-financial-services-with-nvidia-rivermax-and-neio-fastsocket. Accessed: 2026-02-17.

[19] NVIDIA. What is AI inference? https://www.nvidia.com/en-us/glossary/ai-inference/. Accessed: 2026-02-17.

[20] PyTorch. CUDA semantics and memory management. https://docs.pytorch.org/docs/stable/notes/cuda.html. PyTorch Official Documentation. Accessed: 2026-03-04.

[21] PyTorch. DistributedDataParallel — notes. https://docs.pytorch.org/docs/stable/notes/ddp.html. PyTorch Official Documentation. Accessed: 2026-03-01.

[22] PyTorch. torch.distributed — distributed communication package. https://docs.pytorch.org/docs/stable/distributed.html. PyTorch Official Documentation. Accessed: 2026-03-01.

[23] PyTorch. torch.nn.parallel.DistributedDataParallel. https://docs.pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html. PyTorch Official Documentation. Accessed: 2026-03-01.

[24] PyTorch. torch.optim.Adam â PyTorch Documentation. https://docs.pytorch.org/docs/stable/generated/torch.optim.Adam.html. PyTorch Documentation. Accessed: 2026-03-13.

[25] PyTorch. torch.utils.data â PyTorch Documentation. https://docs.pytorch.org/docs/stable/data.html. PyTorch Documentation. Accessed: 2026-03-13.

[26] Dave Salvator. Explaining tokens — the language and currency of AI. https://blogs.nvidia.com/blog/ai-tokens-explained/. NVIDIA Blog. Accessed: 2026-02-17.

[27] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert,

a distilled version of bert: smaller, faster, cheaper and lighter. In *NeurIPS EMC^{2}Workshop*, 2019.

[28] Jaime Sevilla and Anton Troynikov. Could decentralized training solve AI's power problem? https://epoch.ai/blog/could-decentralized-training-solve-ais-power-problem. Epoch AI Blog. Accessed: 2026-02-17.

[29] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E. Gonzalez, and Ion Stoica. Fairness in serving large language models. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2024.

[30] Vikranth Srivatsa, Zijian He, Reyna Abhyankar, Dongming Li, and Yiying Zhang. Preble: Efficient distributed prompt scheduling for LLM serving. *arXiv*, 2024.

[31] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic scheduling for large language model serving. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2024.

[32] Ruixing Zong, Jiapeng Zhang, Zhuo Tang, and Kenli Li. Ibing: An efficient interleaved bidirectional ring all-reduce algorithm for gradient synchronization. *ACM Transactions on Architecture and Code Optimization*, 22(1), 2025. Article 35 (March 2025), 23 pages.