University of Victoria

UNIVERSITY OF VICTORIA

Traffic Pattern Analysis and Comparison of Distributed Deep Learning Models

Student :
Li HE

Instructor :
Jianpin Pan

April 18, 2025

Contents

Abstract

This research investigates the network traffic patterns generated during distributed training of deep neural networks across different hardware architectures and topologies. As deep learning models continue to grow in size and complexity, single-device training becomes increasingly impractical, necessitating distributed training approaches that introduce significant communication overhead. We analyze the communication patterns, bottlenecks, and efficiency tradeoffs between NVIDIA DGX-2 systems and Tensor Processing Units (TPUs) when training various deep learning models, including VGG16, ResNet50, and GPT-3. Our methodology combines real-world experiments on the Alliance of Canada infrastructure with detailed simulations using ASTRA-sim, a distributed machine learning simulator. Results demonstrate significant differences in communication efficiency between platforms, with TPUs exhibiting superior network performance despite DGX-2's computational advantages. We observe that model parallelism generates substantially more communication overhead than data parallelism, particularly for larger models, and that fully-connected layers represent critical bottlenecks in the communication pipeline. These findings provide valuable insights for optimizing distributed training configurations based on model architecture, hardware platform, and network topology, potentially reducing training time and resource utilization for large-scale deep learning workloads.

1 Introduction

Deep neural networks (DNNs) have revolutionized numerous domains, including computer vision, natural language processing, speech recognition, and recommendation systems. Their remarkable capabilities come with significant computational demands, driven by increasingly complex architectures and massive training datasets. Modern state-of-the-art models like GPT-3 [1] contain hundreds of billions of parameters, requiring computational resources far beyond what a single accelerator can efficiently provide.

Distributed deep learning [2, 3, 4] has emerged as the primary solution to this challenge, enabling researchers and engineers to train increasingly sophisticated models by distributing the workload across multiple accelerators, often spanning multiple physical machines. However, this distribution introduces a new dimension of complexity: network communication. When training is distributed, processors must frequently exchange information such as gradients, model parameters, and activations, generating substantial network traffic that can become a performance bottleneck.

The efficiency of distributed training depends on multiple interconnected factors [5, 6, 7]: the choice of parallelization strategy (data parallel, model parallel, or hybrid approaches), the underlying hardware architecture (GPUs, TPUs, or specialized accelerators), the network topology connecting the compute nodes, and the specific characteristics of the model being trained. Optimizing these elements requires a detailed understanding of the resulting traffic patterns and how they impact overall training performance.

Despite the importance of network communication in distributed training, there remains a gap in the literature regarding a detailed analysis of traffic patterns across different hardware platforms and model architectures. Most existing research focuses either on algorithmic improvements to reduce communication overhead or on optimizing specific hardware configurations, without a comprehensive cross-platform comparison.

This project addresses this gap by conducting a systematic analysis of traffic patterns

generated during distributed deep learning training. We compare two leading hardware platforms - NVIDIA DGX-2 systems (https://www.nvidia.com/en-in/data-center/dgx-2/) and Google's Tensor Processing Units (TPUs) (https://en.wikipedia.org/wiki/Tensor_Processing_Unit) - across multiple model architectures and network topologies. Our investigation employs both real-world experiments on the Alliance of Canada high-performance computing infrastructure and detailed simulations using ASTRA-sim, a specialized simulator for distributed AI systems developed by Intel, Meta, and Georgia Tech.

The primary objectives of this research are to:

- Characterize and compare the communication patterns generated by different deep learning models during distributed training.
- Identify performance bottlenecks and inefficiencies in network communication across hardware platforms.
- Evaluate the impact of different parallelization strategies on communication overhead.

Our results reveal significant differences in communication efficiency between hardware platforms and parallelization strategies. We find that while NVIDIA DGX-2 systems excel in raw computational performance, TPUs demonstrate superior communication efficiency, particularly for larger models. Furthermore, our layer-by-layer analysis identifies fully-connected layers as critical bottlenecks in the communication pipeline, especially when using model parallelism.

These findings have important implications for the design and optimization of distributed training systems. By understanding the specific communication requirements of different models and hardware platforms, researchers can make more informed decisions about resource allocation, parallelization strategy, and network topology, potentially reducing training time and resource utilization for large-scale deep learning workloads.

The remainder of this paper is organized as follows: Section 2 provides background on distributed parallel training approaches, Section 3 describes the common network topologies used in distributed training, Section 4 presents the real-world experiments over Alliance of Canada, Section 5 additionally discuss the simulation results over ASTRA-sim, and we concluded in Section 6.

2 Distributed Training Approaches

Model parallelism and data parallelism are two primary strategies for distributing the training of large machine learning models across multiple computing resources, such as GPUs. These approaches are especially crucial in deep learning, where both the models and datasets can be extremely large.

2.1 Data Parallelism

In data parallelism, the dataset is split into smaller batches, with each node assigned a different batch of the training data. Every node will have the same copy of the model and processes its assigned data independently. Data parallelism is relatively simple to

implement and is effective when the model fits entirely within a single device's memory. The training process generally involves the following steps:

1. Each GPU or device loads an identical copy of the model.
2. The dataset is divided into smaller batches, with each device receiving a different portion.
3. Each device performs forward and backward passes independently on its own data batch.
4. The gradients computed by each device are then synchronized using techniques like all-reduce to ensure consistency.
5. Finally, each device updates its model weights using the aggregated gradients.

Data parallelism is well-suited for training on large datasets and scales efficiently with the number of data samples. However, the communication overhead in data parallelism arises during gradient synchronization, especially when the model grows.

2.2 Model Parallelism

In model parallelism, the model is split across multiple devices, and each device is responsible for a different portion of the model, such as specific layers or sets of neurons. Unlike data parallelism, where all devices work simultaneously on different data batches, model parallelism processes data sequentially as it flows through the split model components. Communication occurs as intermediate outputs (activations) are passed between devices. The training process in model parallelism generally follows these steps:

1. The model is divided across multiple devices, either by assigning entire layers to different GPUs or by splitting individual tensors (tensor sharding).
2. Each device receives the same input data or intermediate activations from the previous stage.
3. Computation progresses sequentially through the partitioned model, with activations passed between devices.
4. During the backward pass, gradients are similarly communicated across devices to complete the update process.
5. Each device updates only its own portion of the model parameters.

Model parallelism is especially valuable when the model is too large to fit into the memory of a single GPU. However, its performance can be limited by the communication overhead required to transfer data between different devices.

2.3 Hybrid Approaches

In practice, many systems use hybrid approaches combining both methods, particularly for extremely large models like GPT-3. This hybrid approach helps make better use of memory and reduces communication time, making it easier to train very large models efficiently.

3 Network Topologies for Distributed Training

The efficiency of distributed training depends significantly on the network topology connecting the compute nodes. There are three common topologies: ring, fully connected, and switch-based architectures.

3.1 Ring

In a ring topology, compute nodes are arranged in a closed-loop configuration, where each node is connected to exactly two neighboring nodes—one on either side. Data flows in a unidirectional manner around the ring, enabling sequential communication from node to node. This topology is relatively simple to implement and requires minimal cabling or interconnect infrastructure, making it an attractive choice for environments with constrained resources. The ring topology presents several advantages. It is cost-effective, as

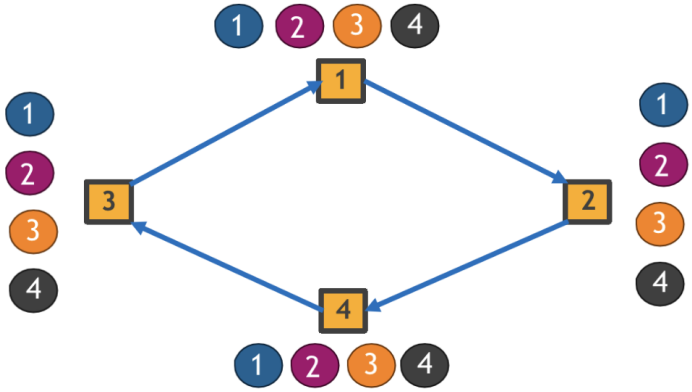

Figure 1: Ring

it requires fewer interconnects compared to more complex topologies such as fully connected networks. Its structural simplicity facilitates ease of deployment and maintenance, particularly in environments with constrained resources. Additionally, the communication paths are fixed, resulting in predictable and deterministic latency. However, this topology also introduces certain limitations. A single node or link failure can disrupt the entire network, scalability is limited as adding nodes requires reconfiguring the ring, and sequential communication can lead to performance bottlenecks in larger systems. Despite these limitations, ring topology is well-suited for small-scale distributed systems with limited resources, where simplicity and cost-effectiveness are more important than achieving high throughput.

3.2 Fully Connected

In a fully connected topology, each compute node is directly linked to every other node in the system, forming a mesh-like network. This architecture enables simultaneous, point-to-point communication between any pair of nodes without relying on intermediaries, ensuring efficient and direct data exchange.

The fully connected topology offers several notable advantages. Its high reliability stems from the presence of redundant communication paths, which reduce the likelihood

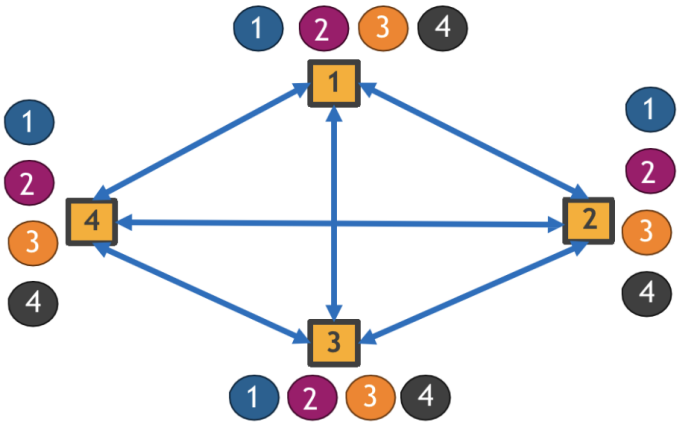

Figure 2: Fully Connected

of network failure. The direct connections between nodes contribute to low communication latency and support high throughput, as multiple data transfers can occur in parallel. However, these benefits come at a significant cost. The topology requires $O(N^2)$ interconnects, making it expensive and impractical for large-scale systems. Additionally, the complexity of configuring and maintaining such a dense network grows rapidly with the number of nodes, posing significant management challenges. Scalability is also limited, as adding new nodes results in an exponential increase in interconnect requirements. Consequently, fully connected topologies are most appropriate for high-performance computing (HPC) clusters where maximum throughput is essential, or for small-scale systems where budget and resource constraints are minimal.

3.3 Switch

Switch-based architectures utilize dedicated switches or routers to interconnect compute nodes. In this configuration, nodes do not communicate directly with each other but instead route data through the switch, which manages and dynamically directs communication paths. This topology offers a high degree of flexibility and scalability, making it suitable for a wide range of system sizes and configurations. Switch-based topologies

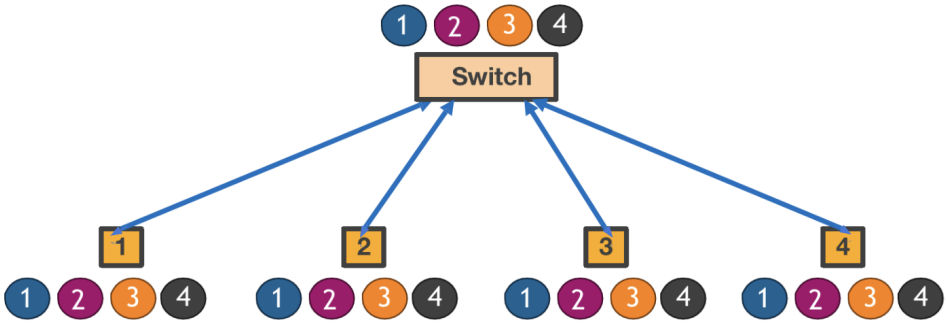

Figure 3: Switch

provide several advantages. They are highly scalable, allowing for the seamless addition of nodes or switches without major architectural changes. Modern switches support high-speed, low-latency communication, contributing to efficient data transfer. Additionally,

these architectures offer fault tolerance through redundant paths and failover mechanisms, enhancing system reliability. From a cost perspective, switch-based networks are generally more economical than fully connected topologies, especially in large-scale deployments. However, there are trade-offs. Performance heavily depends on the capabilities of the switches; inadequate bandwidth or suboptimal configuration can lead to communication bottlenecks. Network design and tuning can also be complex, requiring careful planning to ensure balanced load distribution and minimal latency. Moreover, if redundancy is not properly implemented, switch failure can become a single point of failure. Switch-based topologies are well-suited for large-scale distributed training environments, such as deep learning clusters, and cloud-based infrastructures where dynamic scalability and efficient resource utilization are critical.

4 Real-world Experiments

To gain insight into the real-world distributed training traffic patterns, I began by exploring running distributed training jobs in a real-world platform. Since UVic provides students with the support of computing resource with the platform called Alliance of Canada, I chose it as my experimental platform and successfully ran the distributed training jobs on it in the end.

4.1 Alliance of Canada

The Alliance operates several large supercomputing clusters across Canada – including general-purpose heterogeneous systems like Cedar, Graham, Béluga, and Narval. The clusters are built with fast inter-node interconnects (typically InfiniBand) and uses a Mellanox HDR InfiniBand network (200 Gb/s) to link all nodes. Researchers access these resources through the Slurm workload manager, submitting jobs that request a number of nodes/GPUs and time; the Slurm [1] scheduler then allocates the specified GPUs and coordinates job launch across the cluster. This allows distributed training scripts to run in parallel on multiple GPUs (potentially across several nodes) under a unified job allocation.

4.2 Distributed Training on Alliance HPC Clusters

We conducted distributed training experiments using PyTorch's `DistributedDataParallel` (DDP) framework on the Alliance of Canada HPC clusters. The training script leverages NVIDIA's `NCCL` backend for efficient inter-process GPU communication. Initially, each training process calculates its unique global rank and joins the distributed environment using `torch.distributed.init_process_group`. Training processes are launched using PyTorch's `torch.multiprocessing.spawn` utility, and GPUs are assigned based on each process's local rank.

In our training stage, we use Fashion-MNIST [2] as the dataset for distributed training. Since we are doing the data parallelism, the dataset is loaded and distributed across GPUs using `DistributedSampler`, ensuring that each GPU processes distinct subsets of data. A `WideResNet` architecture—consisting of several convolutional and pooling layers

[1] https://docs.alliancecan.ca/wiki/Using_GPUs_with_Slurm
[2] https://www.kaggle.com/datasets/zalando-research/fashionmnist

with synchronized batch normalization—is employed as the training model. The model is wrapped with PyTorch's DDP to facilitate automatic gradient synchronization across GPUs.

Synchronization points (`dist.barrier`) are strategically placed to coordinate epochs across GPUs, while performance metrics, including epoch timings and throughput, are aggregated using collective operations like `torch.distributed.reduce`. Validation accuracy and loss are averaged globally across processes using `torch.distributed.all_reduce`.

After submitting the training job request, I was assigned to the Cedar cluster with the allocation of 4 NVIDIA V100 GPUs distributed across 2 nodes in which nodes connected via InfiniBand EDR (100 Gbps).

4.3 Numerical Results

The experiments on the Alliance of Canada platform using PyTorch Distributed Data Parallel (DDP) with a Wide ResNet model on the Fashion MNIST dataset produced the following results:

Epoch 1 Summary:

- Time Taken: 81.000 seconds
- Throughput: 740.74 images/second
- Validation Loss: 0.354
- Validation Accuracy: 87.8%
- Communication Time for AllReduce (gradient synchronization): 0.006016 seconds

Epoch 2 Summary:

- Time Taken: 78.931 seconds
- Throughput: 760.16 images/second
- Validation Loss: 0.249
- Validation Accuracy: 91.0% (exceeding the target accuracy of 0.85)
- Communication Time for AllReduce: 0.007944 seconds

These results demonstrate that in a real-world setting with a moderate-sized model and a well-optimized framework like PyTorch DDP, communication overhead constitutes a very small fraction of the total training time (less than 0.01%). This efficiency is partly due to the high-speed InfiniBand interconnect and the relatively small size of the Wide ResNet model. However, these measurements only capture the synchronization time for gradient all-reduce operations and do not provide visibility into the detailed network traffic patterns. For this reason, I turned to ASTRA-sim for more comprehensive analysis.

5 ASTRA-sim Simulator

While I can run my distributed training script on the Alliance Canada clusters, I was facing challenges in developing a method to capture and analyze network performance data. This step is crucial for understanding the traffic patterns and communication overhead in distributed deep learning training.

To cope with the above challenge, I looked into alternative tools and I found the simulator ASTRA-sim (`https://github.com/astra-sim/astra-sim`), which is a distributed machine learning system simulator developed by Intel, Meta, and Georgia Tech. The overview of ASTRA-sim is given in Fig. 4.

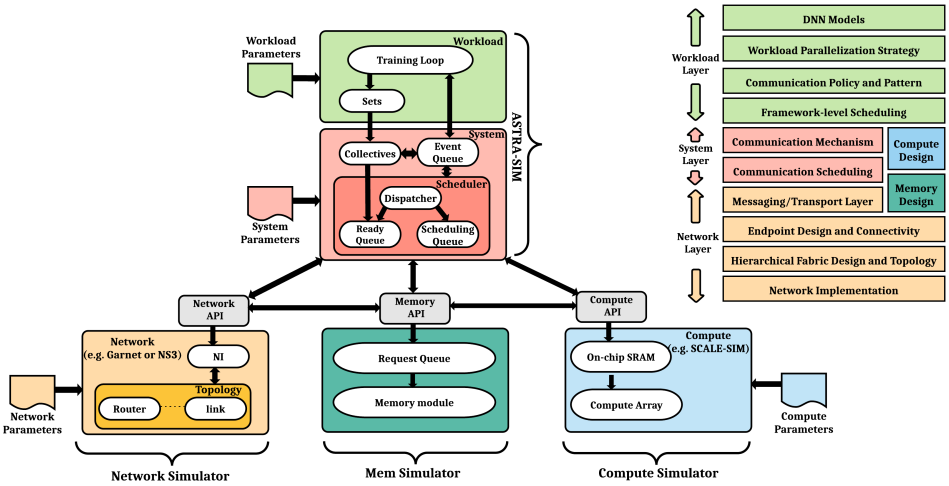

Figure 4: The overview of ASTRA-sim

It is designed to model the complex co-design space of distributed machine learning (ML) platforms, which involve interactions between DNN model architecture, parallelization strategy, scheduling strategy, collective communication algorithm, network topology, and accelerator endpoints. ASTRA-sim enables researchers to systematically study bottlenecks and evaluate futuristic systems at both software and hardware levels for scaling distributed ML.

The integration of the Network Simulator in ASTRA-sim enables us to easily analyze network performance. More importantly, it allows us to customize the topology for the distributed training such as parameter servers, all-connected, ring, etc. This salient feature perfectly matches with my initial goal of traffic pattern study with varying topologies.

5.1 Experimental Settings

5.1.1 System settings:

ASTRA-sim is designed to represent real-world distributed training systems with high fidelity. It provides the parameters of the existing commercialized frameworks:

- NVIDIA DGX-2
- Google Cloud TPU v2

NVIDIA DGX-2 consists of 16 nodes each with 16 V100 GPUs (256 GPUs total). In a DGX-2, GPUs within a node are fully interconnected via NVSwitch (NVLink) and nodes are linked by InfiniBand. ASTRA-sim encodes this as a two-dimensional (2D) network: one dimension represents the intra-node fabric and the second represents the inter-node network. In the DGX-2 case, the first dimension models the NVSwitch connectivity (each GPU has 6 NVLink connections at 25 GB/s each) and the second dimension models the InfiniBand switch connecting the 16 nodes (each with a 100 GbE link, 6.25 GB/s). Similarly, to simulate a Google Cloud TPU v2 pod, which uses a 16×16 2D torus of TPUs (256 chips total) connected by a custom inter-core interconnect (ICI) of 496 Gbps (62 GB/s) per link, ASTRA-sim defines a 2D Ring topology. Each TPU in the simulation is connected in a torus arrangement with two links in each dimension (horizontal and vertical ring), each link having the specified bandwidth (62 GB/s) and latency (e.g. 500 ns). In general, the simulator's network configuration file (.json) explicitly lists: the number of network dimensions, the connectivity type per dimension (e.g. Switch, Ring, Torus, etc.), the number of units in each dimension, and the link latency and bandwidth for each level.

5.1.2 Workload generation:

To simulate a distributed deep learning workload, we need to create the workload configuration (often a .txt file) that lists the sequence of operations (layers and communications) in one iteration of training and input the file into ASTRA-sim. Essentially, the workload file is a structured list of the DNN's layers along with the associated computation and communication requirements. Each layer entry can include several key parameters, typically covering:

- Compute Time: an estimate of how long the forward or backward computation for that layer takes on the given hardware (in cycles or nanoseconds). This is derived from the number of operations and the device's performance. For example, the first convolutional layer of VGG-16 [3] requires 173 million operations, which on a TPUv2 (46 TFLOPS peak) was estimated to take about 3429 ns. If we were simulating the same layer on an NVIDIA V100 GPU (112 TFLOPS), the compute time would be scaled down to 41% (since V100 is 2.43× faster).

- Communication Type: the kind of communication that occurs after this layer's computation, if any. This could be NONE (no communication for that phase) for most forward passes, or a collective operation like ALLREDUCE, ALLGATHER, REDUCE-SCATTER, etc.

- Communication Size: the amount of data to be communicated. This typically corresponds to the size of the layer's parameters or activations that need to be exchanged. In the VGG-16 example, the first conv layer has 1,792 parameters (weights + bias), which is about 3.5 KB of data (assuming 2 bytes per parameter). In a data-parallel setting with gradient averaging, each worker will need to send/receive that 3.5 KB in an All-Reduce for that layer.

In our experiments, we generate the workload configuration files for the following three models:

[3] https://lekhuyen.medium.com/an-overview-of-vgg16-and-nin-models-96e4bf398484

1. VGG16: A convolutional neural network (CNN) architecture developed by the Visual Geometry Group (VGG) at the University of Oxford. It consists of 16 layers, including 13 convolutional layers and 3 fully connected layers. The architecture features a stack of convolutional layers followed by max-pooling layers, with progressively increasing depth.

2. GPT-3: Generative Pre-trained Transformer 3(GPT-3) is a cutting-edge language model developed by OpenAI. It is an autoregressive transformer model featuring 175 billion parameters, making it one of the largest and most powerful language models to date. Due to the large size of the real GPT-3, we only select the first 3 transformer layers to generate the workload.

3. ResNet50: It belongs to the Residual Network (ResNet) family, characterized by its unique use of residual or "skip" connections. These connections allow the model to bypass certain layers, thereby addressing the vanishing gradient problem commonly encountered in very deep networks. ResNet-50 specifically consists of 50 layers, including convolutional layers grouped into four primary stages, each comprising residual blocks.

5.2 Simulation Results

We present a comparative analysis of runtime performance for three deep learning models (VGG16, ResNet50, and GPT-3) executed on two hardware platforms (NVIDIA DGX2 and Google Cloud TPU). The results are organized by model, and within each model we examine individual performance metrics – compute time per layer and communication time per layer.

5.2.1 VGG16

For the **VGG16** model, the data-parallel communication time (`DPCommsTime`) is negligible compared to the model-parallel communication time (`MPCommsTime`). As shown in Fig. 5, on the DGX2 platform, VGG16's `DPCommsTime` is only approximately 48,600 μs, whereas its `MPCommsTime` is around 4,832,700 μs, indicating roughly a 100-fold increase when using model parallelism. A similar trend is observed on the TPU: `DPCommsTime` is about 94,700 μs compared to an `MPCommsTime` of roughly 3,562,700 μs (approximately 37 times greater). This demonstrates that for VGG16, model-parallel communication overhead significantly exceeds data-parallel gradient synchronization cost. TPU's high-speed interconnect slightly reduces MP communication overhead (around 26% lower than DGX2), though its DP overhead is marginally higher.

Fig. 6 illustrates per-layer compute times for each VGG16 layer on DGX2 (blue bars) versus TPU (orange bars). DGX2 executes convolutional layers significantly faster than TPU, particularly notable in deeper layers. For instance, in the last convolutional block (e.g., `block5_conv3`), DGX2's compute time is approximately 80,000 μs compared to TPU's 220,000 μs, nearly three times longer. Final fully-connected layers (`fc1, fc2`) exhibit relatively small compute times on both platforms, underscoring DGX2's superior raw compute efficiency for convolution-heavy workloads.

Communication time per layer for VGG16, depicted in Fig. 7, is minimal initially but escalates significantly in deeper layers, especially in the fully-connected layers `fc1`

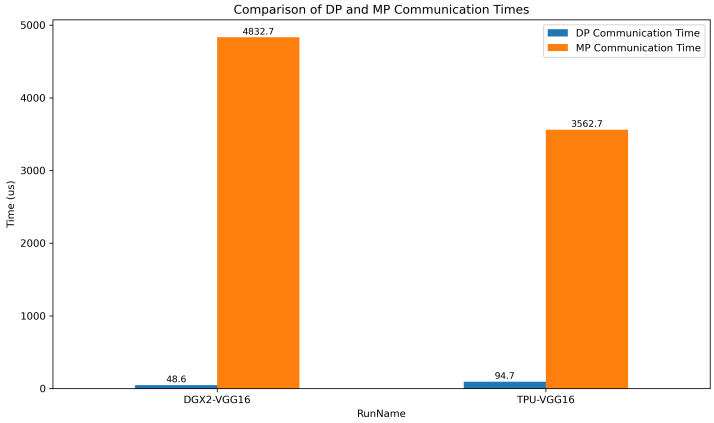

Figure 5: DPCommsTime vs MPCommsTime

and `fc2`. The transition from convolutional blocks to large fully-connected layers incurs substantial data exchanges in model-parallel setups, thus making communication overhead dominant in these final layers.

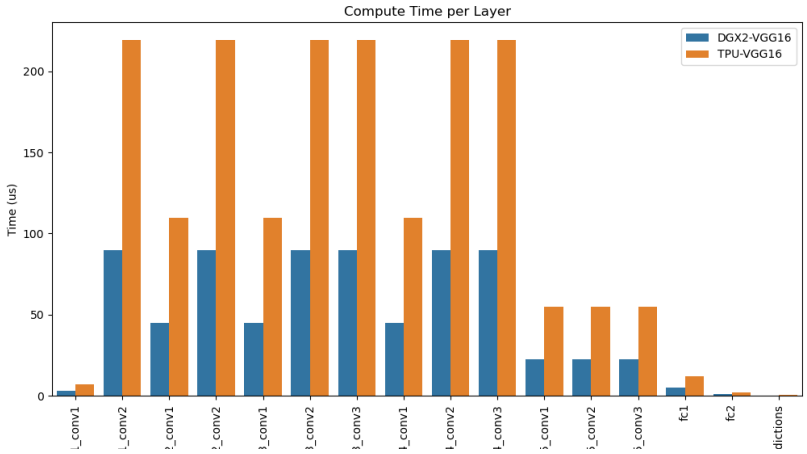

Figure 6: VGG16 Compute Time per Layer

5.2.2 ResNet50

The **ResNet50** model exhibits similar trends between DP and MP communication overheads (Fig. 8). On DGX2, `DPCommsTime` is approximately 53,700 μs, while `MPCommsTime` rises significantly to around 3,223,100 μs (roughly 60 times higher). TPU shows a similar increase, with DP at about 89,800 μs and MP at approximately 3,960,900 μs (about 44 times higher). Notably, the MP communication cost for ResNet50 is lower than VGG16's fully-connected overheads due to smaller activation sizes and final layer dimensions. DGX2's NVLink/NVSwitch interconnect manages MP synchronization slightly faster than TPU in this case.

Fig. 9 displays per-layer compute times for ResNet50, confirming DGX2's consistent advantage over TPU in nearly every convolution layer. Communication time per layer for ResNet50, illustrated in Fig. 10, gradually increases with network depth, becoming prominent in mid to deep layers.

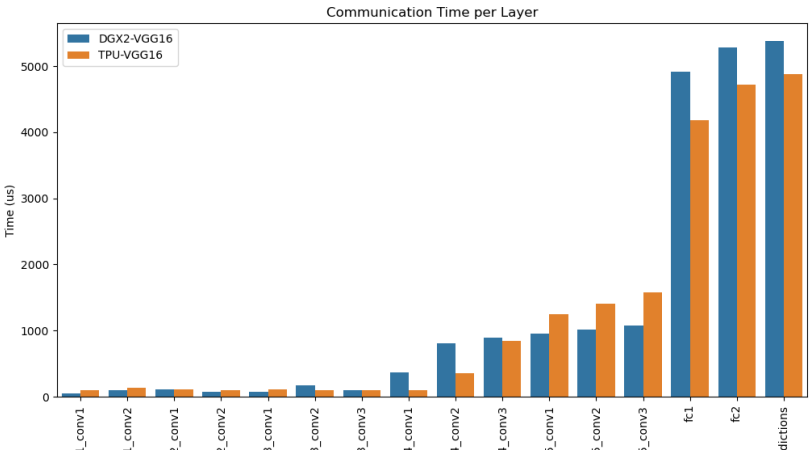

Figure 7: VGG16 Communication Time per Layer

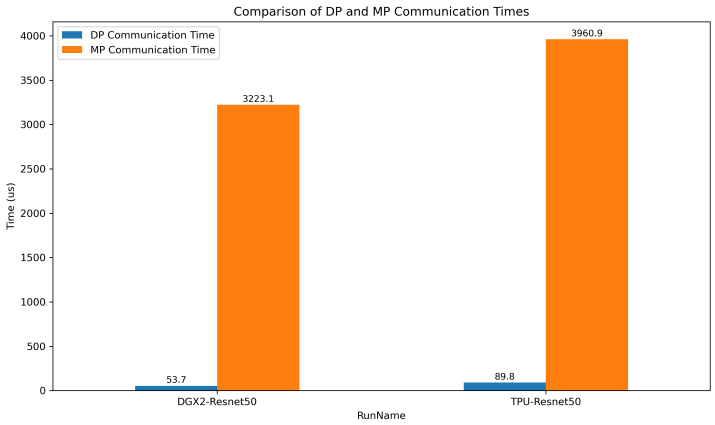

Figure 8: DPCommsTime vs MPCommsTime for RestNet50

Fig. 11 further contrasts compute versus communication times per layer, highlighting that initially, compute dominates (green segments), but in deeper layers, communication (red segments) overtakes compute. This trend is consistent across both platforms, clearly illustrating the eventual dominance of communication overhead in ResNet50.

5.2.3 GPT-3

The **GPT-3** model shows a substantial disparity between DP and MP communication times, as demonstrated in Fig. 12. TPU notably excels in data-parallel communication, significantly outperforming DGX2 by quickly executing gradient synchronization across devices. However, in model-parallel communication, TPU incurs slightly higher overhead (around 250 seconds) compared to DGX2's 206 seconds. This difference likely arises from the parallelization strategy—GPT-3 might be distributed across more TPU cores, increasing the number of communication rounds across the TPU pod network, whereas DGX2 potentially used fewer GPUs with dense NVLink connectivity. Thus, parallelization strategies and hardware topology critically influence GPT-3's communication performance.

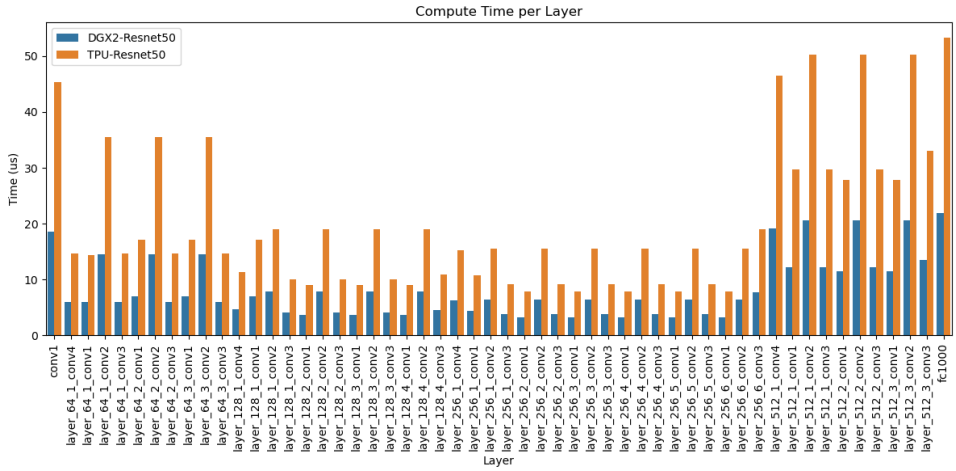

Figure 9: RestNet50 Compute Time per Layer

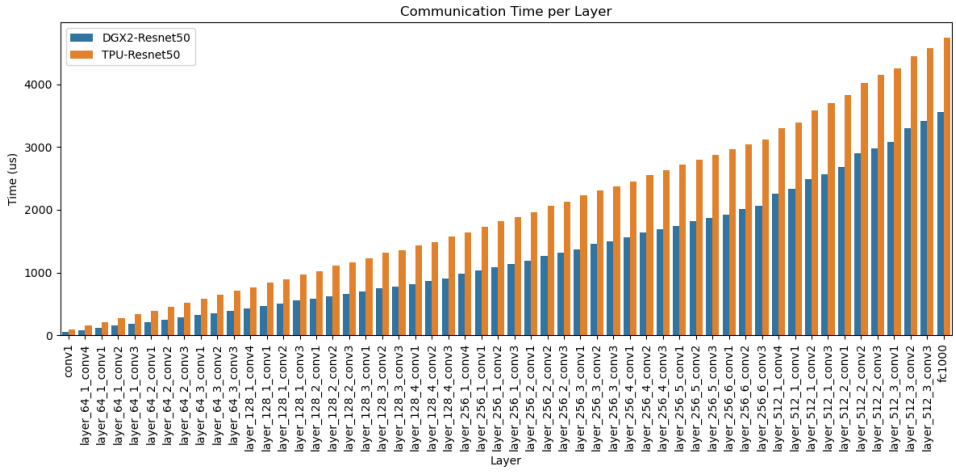

Figure 10: RestNet50 Communication Time per Layer

5.3 Overall Insights

Key insights from the analysis include:

- MP communication dominates runtime in large models (GPT-3, Transformer).
- DP communication is consistently lower than MP.
- DGX2 excels at computation, while TPU shows strength in handling large-scale MP communications.
- Minimizing communication or overlapping it effectively with computation significantly reduces overall runtime.

In conclusion, optimal distributed training performance requires balancing efficient computational power and high-speed interconnects, highlighting the critical role of hardware architecture and communication strategy in scalable deep learning.

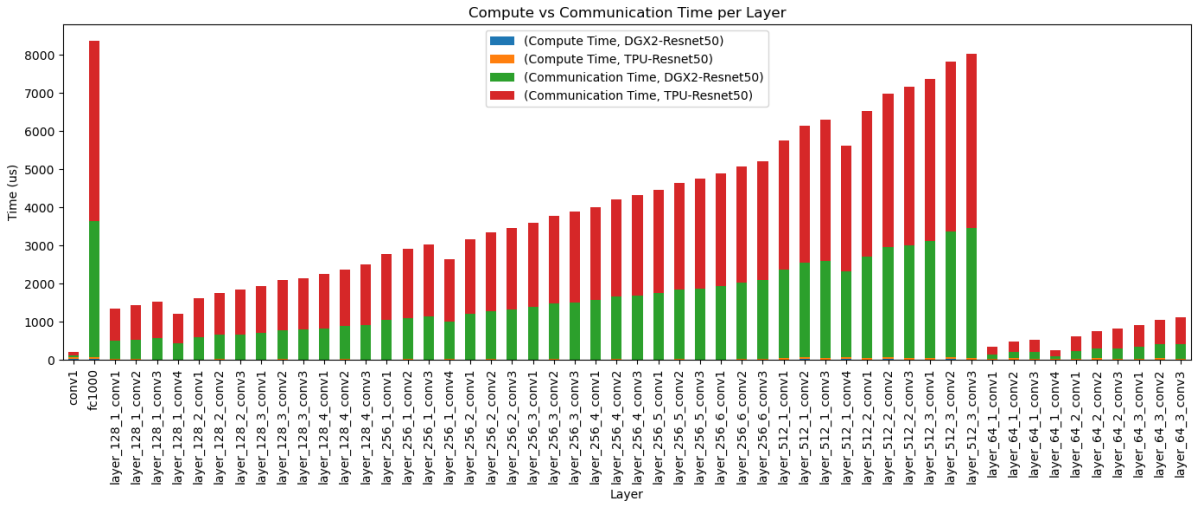

Figure 11: RestNet50 Communication Time VS Compute Time per Layer

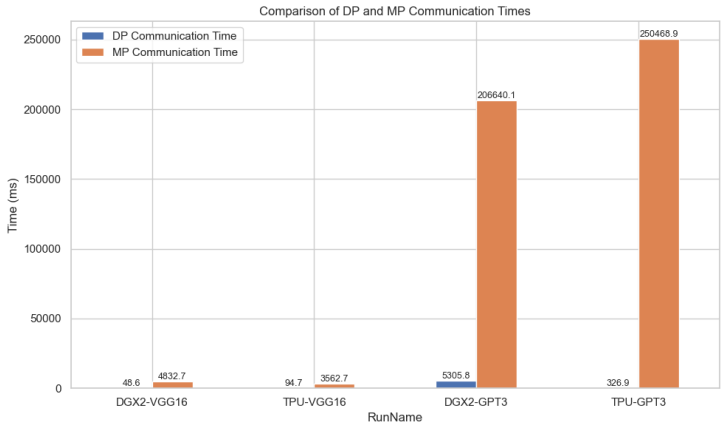

Figure 12: DPCommsTime vs MPCommsTime for GPT3

6 Conclusions and Future Work

6.1 Conclusion

Limitations

Several limitations of this project should be noted:

1. Although ASTRA-sim offers valuable insights into communication patterns, it relies on simplified hardware models that may not fully reflect real-world performance.

2. The analysis was limited to four model architectures, VGG16, GPT-3, RestNet50 and Transformer, which may restrict the generalizability of the findings to other deep learning models.

3. The fast-paced development of hardware platforms suggests that newer GPU and TPU architectures may exhibit communication behaviors that differ from those captured in this project.

6.2 Future Work

To enhance the scope and impact of this study, the following directions will be pursued in future phases of the project:

1. **Expansion of Model Architecture Coverage**

   The analysis will be extended to include a wider variety of deep neural network (DNN) architectures. This includes more transformer-based models (e.g., Vision Transformers, NLP Transformers), recurrent neural networks (RNNs), and hybrid models that integrate convolutional, attention, or memory-augmented components. The goal is to characterize architecture-specific communication patterns—such as gradient synchronization frequency, parameter aggregation strategies, and scalability across topologies—and to identify how these differences influence distributed training performance.

2. **Enhanced Network and Hardware Metric Collection**

   The data collection framework will be expanded to capture a more comprehensive set of system-level metrics, with emphasis on:

   - Bandwidth Utilization: Analyzing how different network topologies (e.g., ring, mesh, hierarchical) affect throughput and latency during collective operations like all-reduce or parameter server communication.
   - Hardware Resource Utilization: Monitoring temporal patterns in GPU/CPU memory usage, cache activity, and compute-idle periods to identify correlations between hardware contention and communication inefficiencies.
   - Batch Size Sensitivity: Conducting controlled experiments with varying batch sizes to assess their impact on communication overhead in both synchronous and asynchronous training scenarios.

3. **Deployment of Real-Time Distributed Training Experiments**

   To validate simulation-based findings under real-world conditions, distributed training jobs will be executed on heterogeneous computing environments, including multi-node GPU clusters and cloud-based platforms. Real-time telemetry tools—such as NVIDIA Nsight Systems and distributed tracing frameworks—will be used to collect detailed performance traces, including synchronization delays and hardware utilization metrics. Insights gained from these experiments will support the development of adaptive optimization techniques, such as topology-aware job scheduling and dynamic batch size adjustment.

Collectively, these efforts aim to provide a deeper understanding of the communication-performance trade-offs in distributed deep learning and to inform the design of scalable, topology-aware training strategies.

References

[1] L. Floridi and M. Chiriatti, "Gpt-3: Its nature, scope, limits, and consequences," *Minds and Machines*, vol. 30, pp. 681–694, 2020.

[2] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li, "Terngrad: Ternary gradients to reduce communication in distributed deep learning," *Advances in neural information processing systems*, vol. 30, 2017.

[3] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang *et al.*, "Large scale distributed deep networks," *Advances in neural information processing systems*, vol. 25, 2012.

[4] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

[5] S. Rajasekaran, M. Ghobadi, and A. Akella, "{CASSINI}:{Network-Aware} job scheduling in machine learning clusters," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024, pp. 1403–1420.

[6] Z. Cai, X. Yan, Y. Wu, K. Ma, J. Cheng, and F. Yu, "Dgcl: An efficient communication library for distributed gnn training," in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, pp. 130–144.

[7] S. Wang, D. Li, and J. Geng, "Geryon: Accelerating distributed cnn training by network-level flow scheduling," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2020, pp. 1678–1687.