ABSTRACT

Title of Dissertation: OPTIMIZING COMMUNICATION IN PARALLEL DEEP LEARNING ON EXASCALE-CLASS MACHINES

Siddharth Singh

Dissertation Directed by: Associate Professor Abhinav Bhatele
Department of Computer Science

Deep learning has made significant advancements across various fields, driven by increasingly larger neural networks and massive datasets. However, these improvements come at the cost of high computational demands, necessitating the use of thousands of GPUs operating in parallel for extreme scale model training. At such scales, the overheads associated with inter-GPU communication become a major bottleneck, severely limiting efficient hardware resource utilization.

This dissertation addresses communication challenges in large-scale parallel training. It develops hybrid parallel algorithms designed to reduce communication overhead, along with asynchronous, message-driven communication methods that enable better overlap of computation and communication. A performance modeling framework is presented to identify communication-minimizing configurations for given workloads. Finally, scalable implementations of latency-optimal collective communication are developed to support efficient training at scale. These contributions improve the performance and scalability of distributed deep learning systems. By

tackling these critical communication challenges, this work contributes to more efficient deep learning training at scale, enabling faster model convergence and better resource utilization across large GPU clusters.

OPTIMIZING COMMUNICATION IN PARALLEL DEEP LEARNING ON EXASCALE-CLASS MACHINES

by

Siddharth Singh

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2025

Advisory Committee:
Associate Professor Abhinav Bhatele, Chair/Advisor
Associate Professor Cristopher Brehm
Professor Tom Goldstein
Assistant Professor Alan Liu
Dr. Olatunji Ruwase

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my advisor, Prof. Bhatele, whose guidance has been instrumental throughout my research journey. Thank you for teaching me how to approach every aspect of research with thoughtfulness and rigor. I am especially grateful for your patience with my time management struggles, your sensitivity to my mental well-being, and the invaluable career advice you've shared along the way. To my labmates, thank you for creating such a supportive and collaborative environment. Your constructive feedback, camaraderie, and readiness for spontaneous happy hours at Looney's made even the toughest days enjoyable. Working alongside you has been an incredible experience. I am sincerely thankful to the DeepSpeed and ADLR teams at Microsoft and NVIDIA for giving me the opportunity to engage with cutting-edge research in the industry. That exposure significantly influenced and helped shape the direction of my dissertation, and I'm deeply appreciative of the experience. I am forever indebted to my parents and family for their unwavering support in all my endeavors. Thank you for instilling in me the value of learning and understanding, and for nurturing a scientific temper from an early age. Your encouragement has been my foundation. To my friends at UMD, thank you for being my home away from home. Your friendship and support have carried me through this journey, and I feel lucky to have had you by my side through it all. Finally, to my best friend, Tiyasha – thank you for being my rock, my biggest cheerleader, and a constant source of strength.

Your belief in me, especially in moments when I doubted myself, meant more than words can say.

GRANTS

I am deeply grateful for the computational resources provided by several high-performance computing centers. These include the Oak Ridge Leadership Computing Facility at Oak Ridge National Laboratory (supported by the U.S. Department of Energy Office of Science under Contract No. DEAC05-00OR22725), where compute time on both the Summit and Frontier supercomputers was essential. Access to Frontier through the DOE's INCITE Program played a critical role in work that culminated in recognition as a 2024 ACM Gordon Bell finalist. Additional resources from the ThetaGPU cluster at the Argonne Leadership Computing Facility (a DOE Office of Science User Facility under Contract No. DE-AC02-06CH11357), the Perlmutter supercomputer at the National Energy Research Scientific Computing Center (NERSC) through awards DDR-ERCAP0022262, DDR-ERCAP0025593, DDR-ERCAP0029890, DDR-ERCAP0029894 and DDR-ERCAP0034262, and the Alps supercomputer at the Swiss National Supercomputing Centre (CSCS) also significantly supported this work. I am sincerely thankful to all these institutions for the generous access to their world-class computational infrastructure, without which this research would not have been possible.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

LLM	Large Language Model
HPC	High Performance Computing
DL	Deep Learning
MoE	Mixture of Experts
ML	Machine Learning

Chapter 1: Introduction

The emphasis on training increasingly large neural networks has gained prominence in deep learning research over recent years (see Figure 1.1). This trend arises from the consistent observation by practitioners that the generalization capability of neural networks improves reliably with an increase in their sizes [11, 12]. Consequently, there has been a noticeable shift towards the development of state-of-the-art AI algorithms relying on neural networks with hundreds of billions of parameters as their foundation [1, 13].

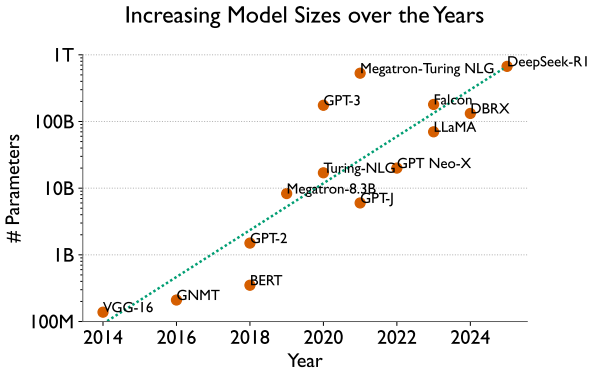

Figure 1.1: Increasing number of parameters in neural networks over the years.

However, this significant increase in the number of parameters is accompanied by an unintended consequence: a substantial increase in the hardware resources required for training these neural networks. The era of training large on a single GPU or even a single node comprising a few GPUs is now a thing of the past. The extensive computational and memory requirements of the training of these multi-billion-parameter models necessitates harnessing hundreds to thousands

of GPUs in parallel.

The primary challenge in scaling parallel or distributed deep learning on multi-GPU supercomputers lies in the substantial communication overhead. While modern GPUs have significantly enhanced computational efficiency—leveraging specialized hardware like Tensor Cores in NVIDIA GPUs—network bandwidth across nodes has not kept pace. As a result, large-scale deep learning frameworks often suffer from inefficiencies due to the high cost of inter-GPU communication.

These communication bottlenecks arise from three key factors. First, distributed deep learning algorithms inherently involve large communication volumes, leading to substantial data transfer overhead. Second, message-passing implementations are often inefficient, with minimal or no overlap between communication and computation, further reducing hardware utilization. Finally, collective communication primitives—critical for distributed training—frequently lack scalability and fail to perform well in latency-sensitive scenarios, particularly when executed across thousands of GPUs. These challenges significantly hinder the efficiency of parallel deep learning frameworks at scale, making communication a major bottleneck in large-scale training.

In light of these challenges, the primary objective of this dissertation is to develop efficient and highly scalable parallel frameworks for training large neural networks. Throughout this dissertation, we investigate communication bottlenecks in various distributed deep learning algorithms and propose solutions to mitigate them. We start with a detailed analysis of the point-to-point communication patterns in pipeline parallelism and suggest a two pronged strategy to reduce these overheads - first, by developing a novel asynchronous message driven communication backend and second, by developing a highly efficient memory optimization strategy that doubles up as a communication optimization. We then study communication bottlenecks in

the context of multi-GPU training of sparse language models, and exploit their sparsity to optimize point-to-point and collective communication in hybrid parallel algorithms. Further, we propose a four-dimensional hybrid parallel algorithm for deep learning, which integrates data parallelism with a three-dimensional tensor parallel approach based on Agarwal's parallel matrix multiplication algorithm [7]. To efficiently identify communication-optimal configurations in the aforementioned algorithm, this dissertation introduces a topology- and placement-aware analytical performance model that rapidly discovers communication-efficient configurations without requiring exhaustive empirical searches across the massive parameter space. Additionally, this dissertation presents highly optimized, latency-sensitive implementations of two critical collective communication operations – all-gather and reduce-scatter – both of which are widely used in distributed deep learning. In latency-sensitive scenarios, such as those involving small message sizes and/or large GPU counts, our implementations substantially outperform the current state-of-the-art, paving the way for more scalable and efficient distributed training of large neural networks.

1.1 Outline of Dissertation

The rest of the dissertation is organized as follows. Chapter 2 provides a survey of the current state-of-the-art works in distributed deep learning. Chapters 3 and 4 deal with optimizing point-to-point and collective communication in a hybrid data and pipeline parallelism framework for training dense and sparse large language models, respectively. Chapter 5 presents a four-dimensional hybrid parallel algorithm for deep learning, along with a performance model to identify communication-minimizing configurations. We use AxoNN, the framework devel-

oped in this chapter, to efficiently scale model training to thousands of GPUs on state-of-the-art multi-GPU supercomputers. Chapter 6 presents a highly scalable implementations of two critical collective operations in parallel deep learning - all-gathers and reduce-scatters. Finally, Chapter 7 studies the optimization of all-to-all and all-reduce communication in large-scale training of mixture-of-expert models.

Chapter 2: Related Work

This chapter presents a survey of current state-of-the-art techniques in distributed deep learning. It first discuss the three types of distributed deep learning algorithms - data, intra-layer or tensor parallelism, and inter-layer or pipeline parallelism. Then it discusses relevant works in the area of optimizing collective communication primitives for traditional HPC and distributed deep learning.

2.1 Data Parallelism

Due to its inherent simiplicity, data parallelism has been the go-to algorithm for parallelizing neural network training. Let us now discuss the various setting in which data parallelism has been studied.

2.1.1 Data Parallelism for Models that Fit on a Single GPU

When training models that fit on a single GPU with data parallelism, the bottleneck in performance is singular - the synchronous all-reduce operation to gather the gradients across all GPUs. This problem is further exacerbated by the increasing computational capabilities of hardware accelerators, which causes a decrease in the computation to communication ratio.

Initial attempts to reduce the communication overhead targeted introducing asynchrony in

the stochastic gradient descent (SGD) algorithm [14–16]. However, Chen et al. [17] demonstrate that synchronous SGD variants converged faster to higher accuracies than their asynchronous counterparts.

Efforts to minimize communication bottlenecks continued. Zhang et al. [18] devise a strategy known as Wait-Free Backpropagation (WFBP) to interleave GPU and CPU computation and communication. WFBP reduces bursts in network traffic and lowers overall network strain. Using WFBP, Zhang et al. achieve speed-ups in training times in 16 and 32 single-GPU machines. WFBP has become the de-facto approach for data parallelism frameworks.

PyTorch DistributedDataParallel (DDP) [19], Horovod [20] and Livermore Big Artificial Neural Network (LBANN) [21] toolkit are three open source frameworks designed to assist in transitioning models into a distributed environment. Out of these frameworks PyTorch DDP has been extremely popular among the deep learning community due to its seamless integration with PyTorch [22]. Horovod is an implementation of WFBP for TensorFlow by Uber. LBANN accelerates parallelized deep learning by taking advantage of high performance computing hardware. These implementations share an uncanny similarity in the way they optimize WFBP. Instead of having an individual all-reduce call for each parameter tensor, they fuse parameter tensors into fixed size bins. All reduce calls are made at the granularity of these fused parameter bins. This increases network bandwidth utilization and thus the overall performance of these frameworks. Although the fused tensor bin-size is kept as a tunable hyperparameter, Li et al. [19] demonstrate that the default bucket size of PyTorch DDP i.e. 25MB is a reasonable choice for efficient scaling.

2.1.2 Data Parallelism for Models that do not fit on a Single GPU

Given the abundance of large training datasets neural networks with increasingly larger number of parameters have led to tremendous gains in performance on a variety of training tasks. As models and datasets grow in size GPU memory capacity becomes a major bottleneck. Data parallelism requires each GPU to store its own copy of the neural network. With larger models and datasets the memory required to house the activations, gradients and parameters of these neural networks often exceeds the capacity of a single GPU DRAM. Data parallelism is thus rendered infeasible for training large models without memory optimizations.

Zero Redundancy Optimizer (ZeRO) [23] is a framework built over PyTorch to reduce per-GPU memory consumption. The paper observes that most memory during training is occupied by optimizer states, gradients, and parameters. ZeRO partitions these model states across GPUs to remove memory redundancies. With ZeRO, memory reduction scales proportionally with the number of GPUs while communication overhead only increases by a constant factor of 1.5x. The paper finds improvements in model size, training performance, and scalability with 100 billion parameter models on up to 400 GPUs using the Adam optimizer [24] and mixed precision. Researchers at Microsoft have used ZeRO to train one of the largest neural networks in language modeling literature: a 17B parameter neural network called the Turing-NLG. Similar ideas to ZeRO have been implemented natively in PyTorch under the Fully Sharded Data Parallelism (FSDP) [25] framework. FSDP is a memory-efficient data parallelism framework that partitions the model across GPUs and shards the optimizer states and gradients across the model partitions.

Out-of core training algorithms like NVIDIA's vDNN [26] are often used to train neural networks on a single GPU with insufficient DRAM capacity. These algorithms move data back

and forth between the CPU and the GPU to free up space on the GPU. KARMA [27] is a framework built over PyTorch that extends this out-of-core approach to data parallelism on multiple GPUs. They design an efficient algorithm for automatic offloading and prefetching of activations and parameters of the neural network to and from the CPU DRAM. These capabilities are further extended to support multi-GPU models by performing weight updates on the CPU. KARMA sees a 1.52x speed-up against other state-of-the-art out-of-core methods. It provides an efficient way to utilize data parallelism for large models that would otherwise necessitate other frameworks. Zero-Infinity [28] is another framework that provides support for out-of-core data parallel training for multi-billion parameter models. Using their memory optimizations, The authors are able to deploy a 32 trillion parameter model on as little as 512 GPUs while maintaining a decent throughput of around 40% of the peak.

2.1.3 Designing ML Algorithms for Large Batch Training

One way to reduce the proportion of time spent in the all-reduce communication in data parallelism is to simply train with larger batches. However, it has been empirically shown that an extremely large effective mini-batch size has an adverse effect on the statistical efficiency of neural network training [29]. To combat this, researchers have been striving to design modified versions of standard optimizers for large batch training. Krizhevsky [30] proposes to scale LR linearly with mini-batchsize. Problems emerge as more workers are added to accelerate training: large LR values result in accuracy losses and training instability. Goyal et al. [29] propose a LR warmup scheme to combat accuracy loss. Training begins with a lower LR that slowly builds up to a target value following the linear scaling rule. The paper was able to train ResNet-50 with a

mini-batch size of 8K and accuracy matching smaller mini-batch models.

You et al. [31, 32] devise Layer-wise Adaptive Rate Scaling (LARS) as an alternate approach to LR warmup. LARS adapts the global LR to create separate LRs per model layer based on the ratio between layer weights and gradient updates. The paper observes this ratio varies across layers and provides insight into the efficacy of a layer's weight updates. You et al. utilize LARS to train AlexNet and ResNet-50 with a mini-batch size of 32K without accuracy loss.

LARS experiences inconsistent performance gains across different deep learning tasks. You et. al [33] propose a general strategy to adapt any iterative optimizer for large mini-batch training. They apply this strategy to create LAMB using the Adam optimizer as a base. Using LAMB, You et al. scale BERT training to a mini-batch size of 32K without performance degradation.

2.2 Intra-Layer or Tensor Parallelism

Tensor parallel algorithms work by parallelizing the computation of every constituent layer of the neural network. Although neural networks are constructed using a plethora of layer types, most frameworks for tensor parallelism focus on fully-connected (FC) or convolution layers. This is because most of the other layer types like activation [34, 35] or norm functions [36–38] are embarrassingly parallel in nature and thus trivial to parallelize. The most widely used tensor parallel framework is Shoeybi et al.'s Megatron-LM [39]. In their work the authors propose an algorithm to parallelize a pair of FC layers. They apply their technique to parallelize large GPT style transformers efficiently within GPUs in a node. Along with a parallel framework, Megatron-LM also open sourced extremely efficient sequential implementations of the transformer layers

as well as data loaders. As a result, their framework has been widely used to train some of the largest language models in existence like Megatron-Turing-NLG-530B [40], Bloom-175B [41], and Turing-NLG [23]. However, their approach becomes inefficient for models that do not fit on a single node [42]. As a result, a number of other works have proposed recently that attempt to alleviate this issue. Qifan et al. propose a 2D tensor parallel algorithm for FC layers [43] based on the SUMMA algorithm for distributed matrix multiplication. Similarly, Wang et al. propose a 2.5D parallel algorithm for FC layers [44]. Zhengda et al. introduce a 3D tensor parallel algorithm based on Agarwal's distributed matrix multiplication [7]. Jangda et al. develop high performance GPU kernels that overlap computation with communication in Megatron-LM's algorithm [45]. Dryden et al. propose channel and filter parallelism for convolution layers [46]. Wang et. al. propose using asynchronous sends instead of all-gather operations for a 2D tensor parallel scheme to overlap communication and computation [47]. Merak [48] introduced an automated framework for 3D parallelism (data + tensor + pipeline) based on graph partition and proxy model graph, along with techniques to overlap communication with computation in pipeline and tensor parallelism. Li et. al. propose Oases that overlaps backward pass communication with activation recomputation [49]. Wang et al. propose to chunk the input batch and overlap the communication and computation of different chunks [50].

2.3 Inter-Layer or Pipeline Parallelism

True inter-layer parallelism can only be achieved by pipelining i.e. having multiple mini-batches active in the system at any given instance. There are two ways to achieve pipelining: with and without flushing. In this section, we discuss the pros and cons of both approaches. We

also provide an overview of frameworks that implement these approaches.

2.3.1 Pipelining with Flushing

Pipelining with flushing divides a mini-batch into micro-batches of equal size. These micro-batches are injected one by one into the system. GPUs accumulate gradients from all the micro-batches in the system. A GPU updates its weights only after it has finished the backward pass of the last micro-batch. The next mini-batch and its corresponding micro-batches are injected after all the GPUs have finished updating their weights. This approach to pipelining is also called micro-batching. The number of micro-batches is usually kept to be much larger than the number of workers so that each worker can compute concurrently. Ensuring optimum hardware utilization requires having a large mini-batch size. To maintain statistical efficiency at large mini-batch sizes the same set of solutions discussed in Section 2.1.3 can be used. Worker GPUs incur idle time between the forward pass of the last micro-batch and the backward pass of the first micro-batch. These are called pipeline bubbles. They reduce the overall hardware utilization of the system. A load balanced mapping of layers to GPUs is absolutely critical to maximize performance. The load balancing algorithm must also be communication-aware. This is because activations and gradients exchanged at GPU boundaries can be in the magnitudes of GBs for large neural networks. An efficient implementation of pipelining with flushing must have load balancing support.

This idea was first introduced by Huang et al. in GPipe [51]. Using GPipe they trained a 557M parameter neural network - AmoebaNet-B [52] on the ImageNet [53] dataset and surpassed the state of the art in a number of downstream image classification tasks. TorchGPipe [54]

is an unofficial open-source implementation of GPipe built on the PyTorch [22] backend. GEMS (GPU-Enabled Memory Aware Model-Parallelism System) [55] introduces a novel approach to increase hardware utilization. This framework proposes an algorithm to train two neural networks concurrently using pipelining without flushing on multiple GPUs. They double the throughput of the system by overlapping the forward and backward passes of the two neural networks. We refer the reader to their paper for the details of their implementation. Recently ZeRO [23] and Megatron [39] also extended support for this approach towards inter-layer parallelism. TorchGPipe [54] provides a load balancing algorithm that seeks to balance the net execution time of the forward and backward pass of a micro-batch on each GPU. However, their algorithm ignores the communication overhead of exchanging tensors across GPU boundaries. Megatron divides the layers of a transformer across GPUs, which is optimal because all the layers of a transformer are identical. ZeRO also provides an identical strategy that divides the layers equally across GPUs. Additionally, they also support a load balancing algorithm that equalizes GPU memory consumption across GPUs. AxoNN [56] introduced a novel asynchronous communication backend for inter-layer parallelism. To the best of our knowledge this is the first work that utilizes asychrony for increasing hardware utilization by opting for MPI instead of NCCL. They also introduce a memory optimization algorithm that they use to decrease the pipeline depth, increase data parallelism and outperform the state-of-art by 15%-25% on models with as many as 100 billion parameters.

2.3.2 Pipelining without Flushing

In this approach the number of mini-batches active in the system is kept constant. As soon as a mini-batch finishes its backward pass on the first GPU a new mini-batch is injected into the system to maintain full pipeline occupancy. Unlike pipelining with flushing, weight updates on a GPU take place as soon as it is done with the backward pass of a mini-batch. This method of pipelining seeks to increase hardware utilization by removing flushing induced bubbles in the pipeline. However, statistical efficiency of such a training algorithm reduces drastically. This is due to a problem called weight staleness that occurs when newer mini-batches in a pipeline encounter stale weights in forward passes which are yet to be updated with the backward pass of older mini-batches. This is one of the major reasons why pipelining without flushing has not seen widespread adoption. PipeDream [57] is a framework that implements pipelining without flushing. It employs an algorithm called weight stashing to counter weight staleness. We refer the reader to their paper for exact details of the implementation. Chen et al. [58] suggest predicting future weights from stale weights using a variant of SGD with momentum [59]. PipeDream additionally proposes a static load balancing algorithm that is communication aware. It instruments each layer and uses the profiling data in its load balancer. Their framework also has an additional provision to replicate compute-intensive layers across GPUs to increase their throughput. These replicated layers synchronize their gradients via all-reduce after each backward pass.

2.4 Optimizing Collective Communication

Optimizing collective communication has been a long-standing challenge in high-performance computing (HPC) and parallel computing and has been the focus on extensive research. Thakur

et al.'s seminal work focuses on optimizing a plethora of collective operations including all-gathers and reduce-scatters in MPICH [60]. The authors explore the design space of several algorithms for each collective, and provide guidelines for selecting the most approporiate algorithm for different scenarios. In contrast, our work focuses on large-scale deep learning workloads and with messages sizes in the tens to hundreds of megabytes. Patarsuk et al. propose the bandwidth-optimized ring algorithm for all-reduce operations [61]. Graham et al. develop optimize MPI collective to effectively exploit shared memory on multi-core systems. Chan et al. [62] develop highly optimized collectives for the IBM Blue Gene/L, exploiting unique properties of the system [63]. Kandalla et al. develop a scalable multi-leader hierarchical algorithm for all-gather [64]. Note that the focus of these works is on optimizing the performance of collective communication in traditional HPC workloads with small message sizes, and not on the large message sizes inherent to deep learning.

De Sensi et al. study the performance of NCCL, RCCL and Cray-MPICH across various state of the art supercomputers across a variety of latency and bandwidth bound scenarios [65]. Cho et al. propose a mutli-level hierarchical ring algorithm for all-reduce and study the tradeoff of bandwidth and latency between the flat and hierarchical ring algorithms [66]. In this work, we build on this and exploit more latency optimal algorithms like recursive doubling/halving in the inter-node levels of the hierarchy and also demonstrate how this design can be utlized to load-balance network traffic across NICs. Note that similar hierarchical designs have been explored in other works as well [67, 68]. Cai et al. develop a systematic theoretical approach to synthesize novel communication algorithms for optimizing collective communication on a particular topology [69]. Cho et al. develop a strategy to maximize the overlap of a tree-based all-reduce with the computation in neural network training [70]. There is also a body of work

focused on exploiting data compression to minimize communication overheads in distributed deep learning. For example, Feng et al. optimize all-to-all communication in recommendation model training via a novel error-bounded compression algorithm [71]. Huang et al. develop hZCCL, a communication library that enables collective operations on compressed data [72]. Zhou et al. develop a GPU-based compression scheme for all-gathers and reduce-scatters [73] and optimize FSDP [25] training at scale.

Chapter 3: An Asynchronous Message Driven Approach for Hybrid Pipeline and Data Parallelism

This chapter presents prior work [56] focused on reducing communication overhead in hybrid parallel training algorithms that combine data and pipeline parallelism. Specifically, it introduces AxoNN, a framework that employs a novel asynchronous, message-driven algorithm to optimize communication efficiency in pipeline parallelism. By leveraging this approach, AxoNN aims to improve scalability and performance in large-scale training workloads.

3.1 Designing a Hybrid Pipeline and Data Parallel Deep Learning Framework

We now present the design of our new framework. AxoNN combines inter-layer parallelism and data parallelism to scale parallel training to a large number of GPUs.

3.1.1 A hybrid approach to parallel training

The central idea behind AxoNN's hybrid parallelization of neural networks is to create a hierarchy of compute resources (GPUs) by dividing them into equally sized groups. Each group of GPUs can be treated as a unit that has a full copy of the network similar to a single GPU in the case of pure data parallelism. Each group works on different shards of a batch concurrently to provide data parallelism. GPUs within each group are used to parallelize the computation

associated with processing a batch shard using inter-layer parallelism. In the case of AxoNN, we arrange GPUs in a virtual 2D grid topology as shown in Figure 3.1. GPUs in each row form a group and are used to implement inter-layer parallelism within each group. The groups together are used to provide data parallelism by processing different shards of a batch in parallel. We use G_{data} and G_{inter} to denote the number of data-parallel groups and the number of GPUs inside each data-parallel group respectively.

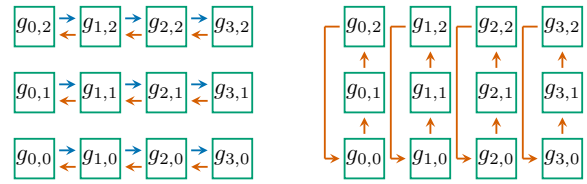

Figure 3.1: AxoNN uses hybrid parallelism that combines inter-layer and data parallelism. In this example, we train a neural network on 12 GPUs in a 4×3 configuration (4-way inter-layer parallelism and 3-way data parallelism). The blue and red arrows represent communication of activations and gradients respectively. In inter-layer parallelism, these gradients are w.r.t. the output activations, whereas in data parallelism, these gradients are w.r.t. the network parameters.

Algorithm 1 AxoNN's hybrid training algorithm for GPU $g^{i,j}$ in a $G_{inter} \times G_{data}$ configuration

```
1:  function TRAIN(neural_network, dataset ...)
2:      nn_shard ← instantiate neural network shard for g^{i,j}
3:      while training has not finished do
4:          next_batch ← get next batch from dataset
5:          batch_shard ← get batch shard for g^{i,j}
6:          DATA_PARALLEL_STEP(nn_shard, batch_shard ...)
7:          run the optimizer
8:      end while
9:  end function
10:
11: function DATA_PARALLEL_STEP(nn_shard, batch_shard ...)
12:     INTER_LAYER_PARALLEL_STEP(nn_shard, batch_shard ...)
13:     All-reduce on nn_shard.∇θ⃗
14: end function
```

Algorithm 1 explains the working of AxoNN's parallel algorithm from the point of view of one of the GPUs $g^{i,j}$ in the 2D virtual grid. Training begins in the TRAIN function (line 1)

which takes a neural network specification and a training dataset as its arguments. For each GPU, we first instantiate a neural network shard (contiguous subset of layers) that GPU $g^{i,j}$ will be responsible for in the inter-layer phase (line 12). In the main training loop (lines 3-7), we divide the input batch into G_{data} shards (line 5) and run the data parallel step on the corresponding shard of $g^{i,j}$. The data parallel step first calls the inter-layer parallel step followed by an all-reduce on the gradients of the network shard. In the optimizer phase, we run a standard optimizer used in deep learning such as Adam [24]. Next, we provide details about the inter-layer and data-parallel phases in AxoNN.

3.1.2 Data parallel phase

We outline AxoNN's data parallel phase in Algorithm 1. The central idea of AxoNN's data parallelism is to divide the computation of a batch by breaking it into G_{data} equal sized shards and assigning each individual shard to a row of GPUs in Figure 3.1 (line 5). Each row of GPUs then initiates the inter-layer parallel phase on their corresponding batch shards (line 12). On completion of the inter-layer parallel phase, GPUs in a column of Figure 3.1 issue an all-reduce on the gradients ($\nabla\vec{\theta}$) of their network shards (line 13). This marks the completion of the data parallel phase, after which we run the optimizer to update the weights (line 7).

3.1.3 Inter-layer parallel phase

The algorithm for the inter-layer parallel phase in AxoNN is described in Algorithm 2. We first divide the batch shard into equal sized microbatches (line 2). The size of each microbatch is a user-defined hyperparameter. We define the $pipeline\_limit$ as the maximum number of mi-

crobatches that can be active in the pipeline. To make sure computation can concurrently happen on all GPUs we first inject $pipeline\_limit$ number of microbatches into the pipeline (lines 4-6) by scheduling their forward passes on each of the first GPUs in a row of Figure 3.1 (line 6). The output of the forward pass is then communicated to the next GPU (line 7). We call lines 3-9 the warmup phase. Once, the pipeline has been initialized with enough microbatches, we enter the steady state of the computation (lines 11-31).

In the steady state, each GPU repeatedly receives messages (line 12) and starts the computation for a forward or backward pass of the network shard depending on if the message is received from a GPU before or after it in its row (lines 15 and 21). If the source is $g^{i-1,j}$ (line 13), a forward pass computation is done using the received message (line 14). We then send the output of the forward pass to GPU $g^{i+1,j}$ (line 19) unless GPU $g^{i,j}$ is the last GPU in the pipeline (line 15). If GPU $g^{i,j}$ is the last GPU in the pipeline, it initiates the backward pass. Similarly if the source of the message is $g^{i+1,j}$ (line 21), the GPU starts the backward pass computation. Once that is complete, we send the output to $g^{i-1,j}$ (line 28) if GPU $g^{i,j}$ is not the first GPU in the pipeline. If it is the first GPU, we inject a new microbatch into the pipeline by initiating its forward pass (lines 24-26). This ensures that the pipeline always has a steady number of microbatches equal to the $pipeline\_limit$ in its steady state. This process repeats until all of the messages for all microbatches of the batch shard have been received and processed (line 11).

Algorithm 2 AxoNN's inter-layer parallelism for GPU $g^{i,j}$ in a $G_{inter} \times G_{data}$ configuration

```
1: function INTER_LAYER_PARALLEL_STEP(nn_shard, batch_shard ...)
2:     microbatches ← divide batch_shard into microbatches
3:     if i = 0 then
4:         for _ in pipeline_limit do
5:             next_microbatch ← microbatches.POP( )
6:             output ← nn_shard.FORWARD(next_microbatch)
7:             SEND(output, g^{i+1,j})
8:         end for
9:     end if
10:
11:    while messages to receive do
12:        msg ← RECEIVE( )
13:        if msg.source = g^{i-1,j} then
14:            output ← nn_shard.FORWARD(msg)
15:            if i = n_inter − 1 then
16:                output ← nn_shard.BACKWARD(1)
17:                SEND(output, g^{i-1,j})
18:            else
19:                SEND(output, g^{i+1,j})
20:            end if
21:        else if msg.source = g^{i+1,j} then
22:            output ← nn_shard.BACKWARD(msg)
23:            if i = 0 then
24:                next_microbatch ← microbatches.POP( )
25:                output ← nn_shard.FORWARD(next_microbatch)
26:                SEND(output, g^{i+1,j})
27:            else
28:                SEND(output, g^{i-1,j})
29:            end if
30:        end if
31:    end while
32: end function
```

3.2 Implementation of AxoNN

In this section, we provide details of the implementation of AxoNN in Python using MPI, NCCL [74], and PyTorch [22]. AxoNN is designed to be run on GPU-based clusters ranging from a single node with multiple GPUs to a large number of multi-GPU nodes. Following the MPI model, AxoNN launches one process to manage each GPU. Each process is responsible

for scheduling communication and computation on its assigned GPU. We use PyTorch for implementing and launching computational kernels on the GPU. AxoNN relies on mixed-precision trainingfor improved hardware utilization [75].

GPUs consume data at a very high rate. As an example, the peak half-precision performance of V100 GPUs on Summit is a staggering 125 Tflop/s. Ensuring that the GPUs constantly have data to compute on requires designing an efficient inter-GPU communication backend, both for inter-layer and data parallelism. We use NVIDIA's GPUDirect technology, which removes redundant copying of data to host memory and thus decreases the latency of inter-GPU communication. We use CUDA-aware MPI for point-to-point communication in the inter-layer parallel phase. In the data parallel phase, we use NCCL for collective communication. We provide explanations for our choices in the sections below.

3.2.1 Inter-layer parallel phase

We first fix the $pipeline\_limit$ to G_{inter} for our implementation of inter-layer parallelism. This ensures that all the GPUs in the pipeline can compute concurrently while placing a low memory overhead for storing activations. When implementing the point-to-point sends and receives in Algorithm 1, we had a choice between CUDA-aware MPI and NVIDIA's NCCL library, both of which invoke GPUDirect for inter-GPU communication. We compared the performance of the two libraries for point-to-point and collective operations on Summit using the OSU Micro-benchmarks v5.8 [2].

Figure 3.2 compares the performance of MPI and NCCL for intra-node (GPUs on the same node) and inter-node (GPUs on different nodes) point-to-point messages (the `osu_latency`

ping pong benchmark). Typical sizes of messages exchanged during point-to-point communication in deep learning workloads are in the range of 1–50 MB. In Figure 3.2, we see nearly identical inter-node latencies. However, MPI clearly outperforms NCCL for intra-node communication. Further, NCCL point-to-point communication primitives block on the communicating GPUs until a handshake is completed. MPI on the other hand allows the user to issue sends/receives without blocking other computation kernels on the GPU. We thus implement AxoNN's asynchronous point-to-point communication backend using MPI.

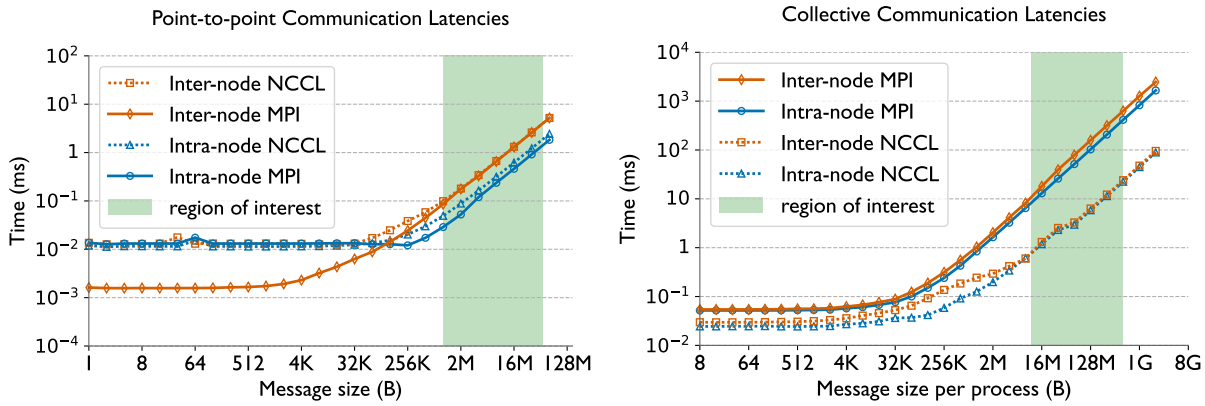

Figure 3.2: Execution time for point-to-point messages (left) and all-reduce operations (right) in MPI and NCCL on Summit using the OSU Micro-benchmarks v5.8 [2]. For point-to-point messages, we use two GPUs on the same and different nodes for the intra- and inter-node experiments respectively. For all-reduce operations, we use six and twelve GPUs for the intra-node and inter-node experiments respectively.

We build our implementation of inter-layer parallelism on top of MPI4Py [76], a library which provides Python bindings for the MPI standard. We use `MPI_Isend` and `MPI_Irecv` to implement the SEND and RECEIVE methods mentioned in Algorithm 1. The `MPI_Irecv` s are issued preemptively at the beginning of a forward or backward pass to overlap the reception of the next message with the computation and thus achieve asynchronous messaging. In lines 13 and 21 of Algorithm 2, we have already shown how AxoNN is designed to support message driven scheduling for inter-layer parallelism. Combined with the asynchronous point-to-point

communication backend discussed in this section, we are able to realize our goal of asynchronous message driven scheduling for improving hardware utilization.

3.2.2 Data parallel phase

We again had a choice between MPI and NCCL for the all-reduce operation in the data-parallel phase and we decided to make that choice based on empirical evidence. Figure **??** presents the performance of the all-reduce operation using MPI and NCCL. In this case, intra-node refers to performing the collective over all six GPUs of a single node and inter-node refers to performing it over 12 GPUs on two nodes. The results demonstrates the significantly better performance of NCCL (dashed lines) over MPI for collective communication. This makes NCCL the clear choice for AxoNN's collective communication backend.

Our implementation of data parallelism uses NCCL for the all-reduce operation over half-precision parameter gradients. We avoid using full-precision gradients to reduce communication times. To prevent overflow in the all-reduce operation, we pre-divide the loss by the total number of microbatches in the input batch. We use PyTorch's `torch.distributed` API [19] for making NCCL all-reduce function calls.

3.3 Memory and Performance Optimizations

In this section, we discuss optimizations that are critical to the memory utilization and performance of AxoNN. We implement these optimizations on top of the basic version of our framework discussed in Section 3.2. We verify their efficacy by conducting several experiments on a 12 billion parameter transformer neural network [8] on 48 NVIDIA V100 GPUs. For all

the experiments in this section, the batch size and sequence length are fixed at 2048 and 512 respectively, and we use mixed-precision training [75] with the Adam optimizer [24].

3.3.1 Memory optimizations for reducing activation memory

Gradient checkpointing reduces the amount of memory used to store activations by only storing the output activations of a subset of layers during the forward pass [77] . For a neural network with N layers $(l_1, l_2, ..l_N)$, this subset of layers is defined as $S_{ckp} = (l_{ac}, l_{2 \cdot ac}, l_{3 \cdot ac} .... l_N)$, where ac is a tunable hyperparameter with the constraint that it should be a factor of N. Activations for layers that are not checkpointed are regenerated during their backward pass. For inter-layer parallelism, it can be shown that the maximum activation memory occupied per GPU with ac as the gradient checkpointing hyperparameter is:

$$\mathrm{M}_{\text{activation}} \propto G_{inter} \times \left(\frac{N}{G_{inter} \times ac} \right) + 1 + ac \tag{3.1}$$

The value $ac = \sqrt{N}$ leads to the lowest value of $\mathrm{M}_{\text{activation}}$. We thus set the value of ac to the factor of $\frac{N}{G_{\text{inter}}}$ (the number of layers on each GPU) closest to $\sqrt{N}$. To the best of our knowledge, our work is the first to derive an optimal value of this hyperparameter for inter-layer parallelism. We implement gradient checkpointing using the `torch.utils.checkpoint` API provided by PyTorch [22].

3.3.2 Memory optimizations for reducing G_{inter} and improving performance of the inter-layer parallel phase

Narayanan et al. show empirically that the performance of inter-layer parallelism deteriorates with increasing values of G_{inter} [42]. They attribute this to an increase in the idle time in the warmup phase with increasing G_{inter}. Below, we show analytically that it is not only the warmup phase, but the entire inter-layer parallel phase which suffers from inefficiency with increasing G_{inter} by virtue of rising communication to computation ratios.

Lemma 3.3.1 Given a neural network, batch size, and number of GPUs, the total amount of communication per GPU in the inter-layer parallel phase is proportional to G_{inter}.

Proof: The total amount of input data a GPU computes on reduces as the number of GPUs used for data parallelism increases. Hence, the amount of data per GPU is inversely proportional to G_{data} given a fixed total number of GPUs that are split between data and inter-layer parallelism. It also follows that the amount of data computed on per GPU is directly proportional to G_{inter}. Assuming that the output activations of each layer of the neural network have the same size, the amount of point-to-point communication per GPU only depends on the amount of input data it consumes and not on the number of layers it houses. Thus the total amount of communication is directly proportional to G_{inter}.

Lemma 3.3.2 Given a neural network, batch size, and number of GPUs, the total amount of computation per GPU in the inter-layer parallel phase is independent of G_{inter}.

Proof: We have shown in Lemma 3.3.1 that the total amount of input data a GPU computes on is

directly proportional to G_{inter}. Since layers are sharded evenly among the G_{inter} GPUs of a row in Figure 3.1, the total amount of computation per instance of the input is inversely proportional to G_{inter}. Thus the total amount of computation per GPU is independent of G_{inter}.

Theorem 3.3.3 Given a neural network, batch size, and number of GPUs, the ratio of the total amount of computation to communication in the inter-layer parallel phase is inversely proportional to G_{inter}.

Proof: Directly follows from Lemmas 3.3.1 and 3.3.2.

We thus expect the efficiency of inter-layer parallelism to decrease with increasing G_{inter} due to an increase in the ratio of communication to computation. We empirically verify this phenomenon by studying the effect of varying G_{inter} on performance. We gather the time spent in the inter-layer parallel phase for processing a batch of input using our reference transformer neural network for values of $G_{inter} = [6, 12, 24, 48]$. The corresponding value of G_{data} is automatically set to be $\frac{48}{G_{inter}}$. We fix the microbatch size and batch size to 1 and 2048 respectively. We also remove the optimizer states so that we do not run out of memory for lower values of G_{inter}. Figure 3.3 illustrates the results of our experiment. As expected, we observe significant gains in performance with decreasing G_{inter}.

Theorem 3.3.3 thus provides a motivation to optimize the implementation by reducing the number of GPUs used for inter-layer parallelism. However, a smaller value of G_{inter} requires the entire network to fit on a smaller number of GPUs, which increases the number of parameters per GPU. This increases the memory requirements per GPU. Lets say the number of parameters per GPU is $\phi = |\vec{\theta}|$ and we are using the Adam optimizer [24] which stores two state variables per parameter. The memory required to store model parameters, gradients and the optimizer states

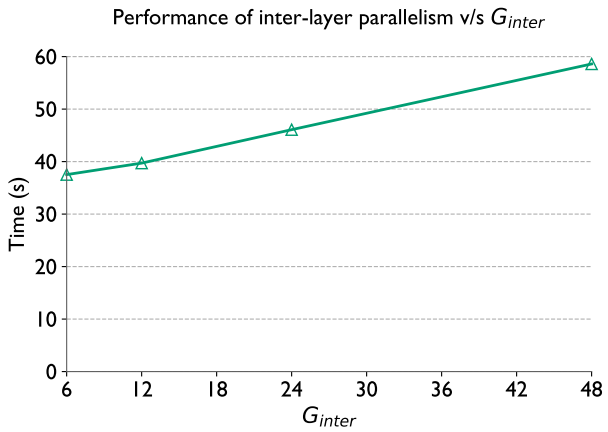

Figure 3.3: Time spent in the inter-layer parallel phase for a single batch for different values of G_{inter} when training a 12 billion parameter transformer model on 48 GPUs of Summit.

comes out to be 20ϕ (4ϕ bytes each for $\vec{\theta}$ and $\nabla\vec{\theta}$, 2ϕ bytes each for $\vec{\theta_{16}}$ and $\nabla\vec{\theta_{16}}$, and 8ϕ bytes for $\vec{s_{opt}}$). Note that this analysis does not include memory required fori storing activations.

At $G_{inter} = 6$ on our reference transformer, this would amount to 40 GB memory per GPU which is 2.5 times more than the 16 GB DRAM capacity of Summit's V100 GPUs. To solve this problem, we introduce a novel memory optimization algorithm that reduces the amount of memory required to store the model parameters and optimizer states by five times.

Implementation: In our memory optimization, only the half precision model parameters ($\vec{\theta_{16}}$) and gradients ($\nabla\vec{\theta_{16}}$) reside on the GPU. Everything else is either moved to the CPU ($\vec{s_{opt}}$ and $\vec{\theta}$) or deleted entirely ($\nabla\vec{\theta}$) before the training begins. The training procedure requires $\vec{s_{opt}}$ and $\vec{\theta}$ on the GPU in the optimizer phase. We save memory by not fetching the entire $\vec{\theta}$ and $\vec{s_{opt}}$ arrays to the GPU, but only small equal sized chunks at a time. We call these chunks buckets and their size as the bucket-size ($bsize$). After fetching a bucket of $\vec{s_{opt}}$ and $\vec{\theta}$, we run the optimizer step on this data and offload it back to the CPU. GPU Memory is saved by reusing buffers across buckets.

The total memory footprint of the optimizer is only $16bsize$ now ($4bsize$ and $8bsize$ for the $\vec{\theta}$ and $\vec{s_{opt}}$ buckets respectively and another $4bsize$ for descaling the half precision gradients).

With our memory optimizations, the total memory requirements to store model parameters and optimizier states is now $4\phi + 16bsize$, down from 20ϕ. As $bsize \ll \phi$, this amounts to a $5\times$ saving in memory utilization. Since the activation memory is unaffected the total memory saved should obviously be less than this number. With $G_{inter} = 24$, $G_{data} = 2$, microbatch size 1 and $bsize = 16$ million our total memory usage for the reference transformer reduces four fold in practice - from 520 GB to 130.24 GB.

Next, we use our memory optimization algorithm with $bsize = 16$ million to reduce G_{inter} from 24 to 6, and study the performance implications. We expect an increase in the time it takes to complete the data parallel phase, because both the amount of data and number of GPUs participating in the all-reduce increases four fold. Figure 3.4 compares AxoNN's performance with and without the memory optimizations. We notice an improvement of 13 percent in the absolute batch timings. While the time for the data parallel phase increases from 0.62s to 4.32s, the corresponding performance gain in the pipelining phase (46.08s v/s 34.05s) compensates for this increase. We expect higher speedups for larger values of batch size, which are typical in large scale model training.

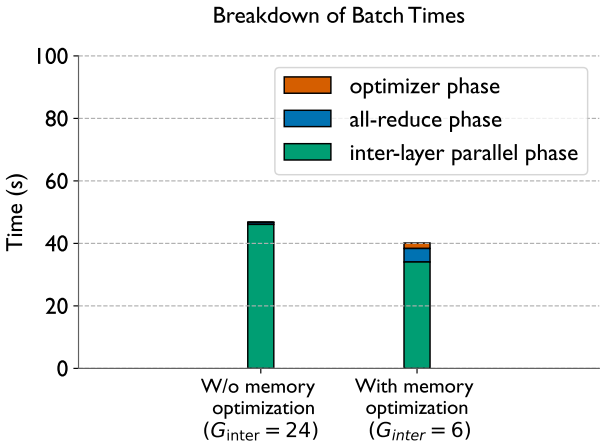

Figure 3.4: AxoNN's performance for a single batch with and without our memory optimization on a 12 billion parameter transformer on 48 GPUs

3.3.3 Overlapping all-reduce & optimizer phases for performance

After optimizing the inter-layer parallel phase, we turned our attention to the less time consuming all-reduce and optimizer phases. We observed that the all-reduce phase (in blue) takes 2.5 times longer than the optimizer phase (in green) (right bar in the Figure 3.4 plot). We hypothesize that by interleaving their executions, we could overlap data movement between the CPU and the GPU in the optimizer phase with the expensive collective communication of the data parallel phase. We explain our approach for enabling this overlap below.

Implementation: The main idea here is to issue the all-reduce call into smaller operations over chunks of the half precision gradients ($\nabla\vec{\theta_{16}}$). For convenience, we keep the size of the chunk as $k \times bsize$, where we call k as the all-reduce coarsening factor. As soon as an all-reduce on a chunk finishes, we enqueue the optimizer step for the corresponding k buckets and start the all-reduce of the next chunk. The key to achieving overlap is to use separate CUDA streams for the optimizer and the all-reduce. Figure 3.5 shows an Nvidia Nsight Sytems profile of our implementation.

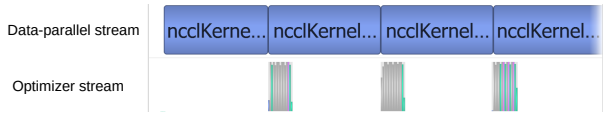

Figure 3.5: An Nsight profile of AxoNN training a 12 billion parameter transformer model on 48 GPUs shows the interleaving of the all-reduce and optimizer phases for a single batch. The two rows represent separate CUDA streams for the optimizer and all-reduce.

We study the variation of the time it takes to finish the combined data parallel and optimizer phases with k in Figure 3.6. At $k = 1$, we observe high overheads due to too many all-reduce calls. Infact, performance is even worse than the case where we had no overlap between the two phases. We observe optimum behavior at two and four. Beyond that, we encounter increasing latencies since increasing k makes the algorithm gravitate towards sequential behavior.

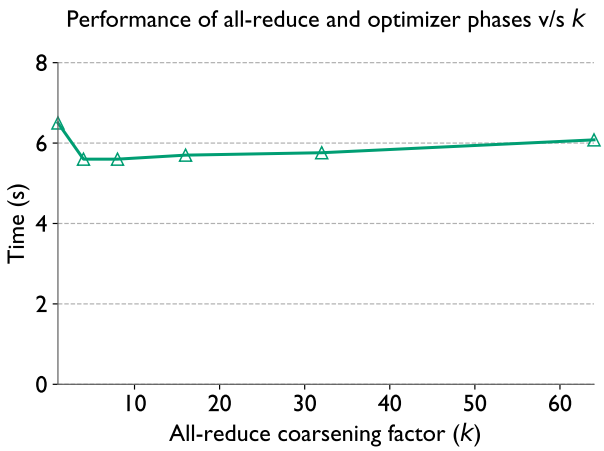

Figure 3.6: Combined execution time of optimizer and all-reduce phases for a single batch versus the coarsening factor, k, for the all-reduce.

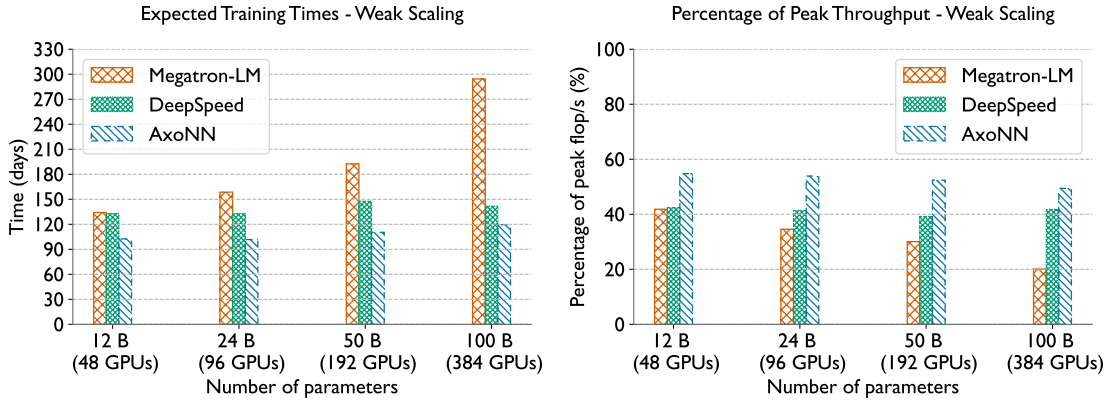

Figure 3.7: Plots comparing the weak scaling performance of AxoNN with other frameworks: expected training times (left) and % of peak GPU throughput (right).

3.4 Results

We now present the results of the experiments outlined in the previous section.

3.4.1 Training validation

It is critical to ensure that parallelizing the training process does not adversely impact its convergence. Diverging training loss curves are a sign of undetected bugs in the implementation or statistical inefficiency of the parallel algorithm. To validate the accuracy of our parallel im-

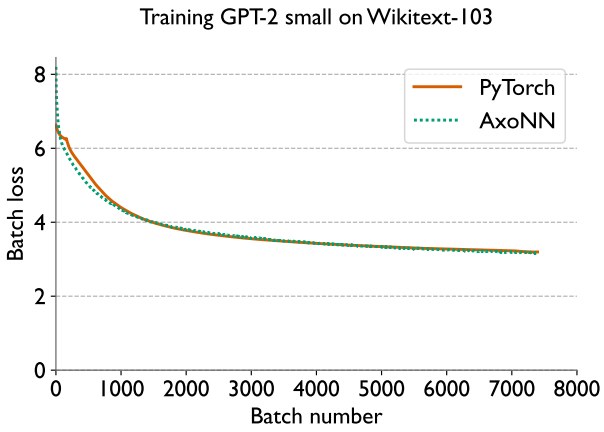

Figure 3.8: Loss curves for training GPT-2 small on the wikitext-103 dataset. We run AxoNN on 12 GPUs ($G_{inter} = 2$) and PyTorch on a single GPU.

plementation, we train the 110 million parameter GPT-2 small to completion using PyTorch on a single GPU and using AxoNN on 12 GPUs with $G_{inter} = 2$. Figure 7.7 shows the training loss for PyTorch, and AxoNN and we can see that the loss curves are identical. This validates our AxoNN implementation.

3.4.2 Weak scaling performance

To be fair to each framework, we tune various hyperparameters for each framework on each GPU count and use the best values for reporting performance results. Table 3.1 lists the optimal hyperparameters we obtain in our tuning experiments for each framework in the weak scaling experiment. Across all model sizes, AxoNN uses four to eight times the number of GPUs for data parallelism as compared to Megatron-LM. This number is identical for AxoNN and DeepSpeed for the 12 billion and 24 billion parameter models but for the larger 50 and 100 billion parameter models, AxoNN uses twice as many GPUs for data parallelism as DeepSpeed. Since data parallelism is embarrassingly parallel, this ends up substantially improving AxoNN's performance.

Table 3.1: Optimal hyperparameter values obtained from tuning experiments for the weak scaling studies.

No. of Params. (billions)	Framework	Micro Batch Size	G_{intra}	G_{inter}	G_{data}
12	AxoNN	8	-	6	8
	DeepSpeed	2	3	2	8
	Megatron-LM	8	3	16	1
24	AxoNN	4	-	12	8
	DeepSpeed	2	3	4	8
	Megatron-LM	1	3	16	2
50	AxoNN	4	-	24	8
	DeepSpeed	1	3	16	4
	Megatron-LM	8	6	32	1
100	AxoNN	2	-	48	8
	DeepSpeed	1	3	32	4
	Megatron-LM	4	12	32	1

Figure 3.7 (left) presents a performance comparison of the three frameworks in the weak scaling experiment. When compared with the next best framework - DeepSpeed, AxoNN decreases the estimated training time by over a month for the 12, 24 and 50 billion parameter models and 22 days for the 100 billion parameter model. For the 100 billion parameter model, AxoNN is faster than DeepSpeed by 1.18$\times$ and Megatron-LM by 2.46$\times$! This is significant for deep learning research as it allows us to train larger models faster. Even at identical values of G_{data} for the 12 and 24 billion parameter models, AxoNN surpasses DeepSpeed because of our asynchronous, message-driven implementation of inter-layer parallelism. These results suggest that AxoNN could scale to training trillion parameter neural networks on thousands of GPUs in the future. AxoNN also delivers an impressive 49-54% of peak half precision throughput on Summit GPUs, outperforming DeepSpeed (39-42%) and Megatron-LM (21-41%) (see Figure 3.7, right).

Chapter 4: Exploiting Sparsity in Pruned Neural Networks to Optimize Large Model Training

Deep learning researchers have developed a variety of pruning algorithms capable of removing (i.e., setting to zero) 80–90% of a neural network's parameters, yielding sparse subnetworks that match the accuracy of their unpruned counterparts. In theory, these sparse subnetworks require significantly fewer floating point operations compared to dense networks. Several sparse matrix multiplication kernels for GPUs have been designed to optimize performance based on the specific sparsity patterns of these subnetworks [3, 78, 79]. However, despite these advancements, existing sparse implementations remain significantly slower than cuBLAS, a widely used library for dense matrix multiplications on NVIDIA GPUs, which underpins deep learning frameworks such as PyTorch and TensorFlow.

Figure 4.1 demonstrates that computing a fully connected layer with 90% sparsity using cuBLAS—where zeros are explicitly retained in the dense matrix representation—is 6–22× faster than Sputnik [3], a state-of-the-art sparse matrix multiplication library for deep learning workloads. This finding suggests that leveraging sparse matrix libraries for improving training performance remains impractical in their current form.

This chapter explores a novel approach [80] that exploits sparse subnetworks to optimize memory utilization and communication in two widely used parallel deep learning algorithms:

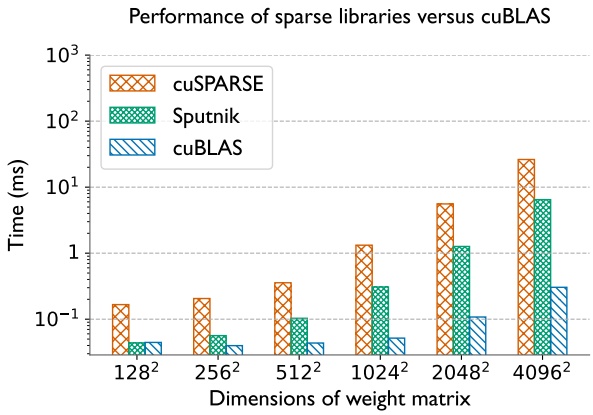

Figure 4.1: Comparison of the execution times of a fully connected (FC) layer with a randomly generated, 90% sparse, square weight matrix in mixed precision. FC layers compute a linear transformation of their input and are a critical component of various neural network architectures, such as transformers [1]. For dense GPU kernels, NVIDIA's cuBLAS is used, while for sparse GPU kernels, NVIDIA's cuSPARSE and Sputnik [3] are employed. The input batch size is fixed at 576, and the size of the weight matrix is varied from 128^2 to 4096^2.

data parallelism and inter-layer parallelism. This approach is integrated into AxoNN, the framework introduced in [56] and discussed in Chapter 3.

4.1 Sparsity-aware Memory Optimization

In this section, we discuss our approach to exploit sparse networks generated by pruning methods to significantly reduce the memory consumption of large model training. We refer to our approach as Sparsity-aware Memory Optimization (SAMO). We discuss SAMO in the context of mixed-precision training [75], which is the predominant mode used for the training of large multi billion-parameter models [23, 40, 41]. However, the optimizations discussed below are general and can also be applied to single-precision training.

Mixed-precision training involves storing the model parameters and gradients in both 16-bit (half-precision) and 32-bit (single-precision), and the optimizer states in 32-bit. The expensive forward and the backward pass are computed in 16-bit for efficiency, whereas the relatively

cheaper optimizer step is done in 32-bit for accuracy. For more details, we refer the reader to Micikevicius et al. [75].

Model parameters, gradients and optimizer states are collectively referred to as the model state [23]. While mixed-precision is compute efficient, storing parameters and gradients in two precisions results in significantly high memory consumption [23] (25% more than single-precision training). For example, in the case of the widely used GPT-3 [1], this adds up to a significant 3.5 TB. For comparison, the DRAM capacity of a single V100 GPU on Summit is a mere 16GB.

Before discussing the details of our approach, we define certain variables as follows:

- θ^{16} and θ^{32} – Network parameters in 16- and 32-bit representation respectively
- $\nabla\theta^{16}$ and $\nabla\theta^{32}$ – Network gradients in 16- and 32-bit representation respectively
- os – 32-bit optimizer states for the network
- $\mathrm{ind} = \bigcup_i \mathrm{ind}_i$ – output of a parameter pruning algorithm, where ind_i stores the indices of the unpruned (non-zero) parameters for the ith layer.

Now, we present how SAMO can help us in significantly reducing the model state memory requirements. Note that SAMO can be applied only after a neural network has been sparsified using a pruning algorithm.

4.1.1 Performance-preserving model state compression

We have already seen in Figure 4.1 that computing the forward and backward passes with compressed sparse parameter tensors on GPUs is not a feasible approach. Thus, a memory op-

timization that tries to compress model states will be efficient only if it is able to utilize dense computation kernels on the GPU. Two important observations about the training process drive the design of our memory optimizations. First, most of the compute in neural network training happens in the forward and the backward pass. Second, out of the various model state tensors discussed previously, the forward and backward passes exclusively use θ^{16} for computation. Thus, we do not compress θ^{16}. This allows us to directly invoke dense computation kernels on GPUs. For saving memory, we compress the other model states i.e., θ^{32}, $\nabla\theta^{16}$, $\nabla\theta^{32}$, and os, which together still comprise 90% of the model state memory, even without θ^{16}! By keeping θ^{16} in an uncompressed format, we thus tradeoff a small proportion of the maximum possible memory savings to gain efficiency in compute.

4.1.2 Implementation of compressed storage

To compress a model state, we convert it to a sparse coordinate (COO) format using the indices of the unpruned parameters (i.e. ind) output by the pruning algorithm. However, being 32-bit (32-bit is sufficient for storing the indices of even the largest models in existence) integers, ind occupies a non-trivial amount of GPU memory. We tackle this issue in two ways. First, we note that all of the model state tensors have zeros at the same indices. Therefore, in our storage scheme, the various COO tensors (i.e. θ^{32}, $\nabla\theta^{16}$, $\nabla\theta^{32}$, and os) share a common index tensor of non-zero values. Secondly, we convert the index tensors of any layer to those of a hypothetical one-dimensional view. As an example, say the non-zero indices for a 2×2 state tensor are $[(0,0),(1,1)]$. In a one dimensional view of the same state tensor (i.e. 4×1), the non-zero values are at indices 0 and 3. Thus, we can save memory by storing only 2 integers (i.e. $[0,3]$), without

any loss of information. In general, for an N-dimensional state tensor, this saves us N$\times$ memory. Having discussed how the various model states are stored by SAMO to optimize for memory, let us now look at how we compute a batch of data with this storage schema.

4.1.3 Training with SAMO

The computation of a batch in neural network training can be divided into three phases - the forward pass, the backward pass and the optimizer step. The forward pass computes the batch loss, the backward pass computes the gradients of the parameters w.r.t. the batch loss, and the optimizer step updates the parameters. Let us now look at how these phases are computed efficiently using SAMO.

Forward Pass: The forward pass of a neural network is done using the half-precision parameters, θ^{16}. As discussed in Section 4.1.1, we store θ^{16} in an uncompressed format with zeros explicitly filled in for pruned parameters. This allows us to exploit efficient dense computation kernels for GPUs, like those available in cuBLAS and cuDNN. Thus, the forward pass with SAMO is exactly the same as that in normal mixed precision training without SAMO.

Backward Pass: The backward pass also uses θ^{16} to compute the batch gradients. Therefore, just like the forward pass, we are able to directly invoke efficient dense computation kernels. However, in Section 4.1.1, we discussed that we store the half-precision gradients in a compressed state i.e. only for the unpruned parameters. Thus, we modify the backward pass to compress the gradients as soon as they are produced for any layer. We do this at the granularity of a layer, and not the entire model, so that we never have to store the uncompressed gradients for the entire model on the GPU memory.

Optimizer Step: In mixed precision training, the optimizer step consists of three element wise operations. The first step involves upscaling $\nabla\theta^{16}$ to $\nabla\theta^{32}$. The second step is running the optimizer using the upscaled gradients $\nabla\theta^{32}$ and the optimizer states, os to update the 32-bit parameters, θ^{32}. The final step is to downscale θ^{32} to θ^{16}. Let us now see how these three steps are done with SAMO.

We do the first step of upscaling $\nabla\theta^{16}$ to $\nabla\theta^{32}$ directly on the compressed tensors itself (as the values for the pruned parameters are always zero) using dense computation kernels. Again due to the same reason, the second step of running the optimizer can be directly computed on the compressed state tensors using dense kernels. This yields the updated parameters in 32-bit i.e., θ^{32}. The final step of downcasting θ^{32} to θ^{16} is not straightforward because these tensors are in a compressed and uncompressed state respectively. To solve this, we first define a new operation,"expansion", as the inverse operation of compression. Essentially, it takes a compressed tensor and the indices of the non-zero parameters to output the uncompressed version. Now, we do the parameter down-casting in three steps. First, we delete the now old uncompressed θ^{16} from the GPU memory. Then we make a copy of θ^{32} in 16-bit. Note that this is essentially the compressed version of our 16-bit parameters. Finally, we "expand" this copy using ind to obtain the updated θ^{16}. Thus, the only modification to the optimizer step is an "expand" operation in the down-casting step.

4.1.4 Analytical model of memory savings

In this section, we derive the memory savings as a result of storing model states with SAMO. We assume that the optimizer of choice is Adam [24], which is the go-to optimizer in

deep learning for large model training. Adam stores two optimizer states per parameter. However, SAMO can be easily extended to work with other optimizers as well.

First let us derive the model state memory consumption without pruning. Let ϕ be the total number of parameters in the neural network before pruning. Now, θ^{16} and $\nabla\theta^{16}$ take up 2ϕ bytes each, whereas θ^{32} and $\nabla\theta^{32}$ take up 4ϕ bytes each. Finally, os, which are stored in single precision take up 8ϕ bytes. This adds up to a total of 20ϕ bytes (2+2+4+4+8). Let us call this quantity $M^{default}$.

Now, let us assume that we are uniformly pruning p fraction of the parameters before applying SAMO. This leaves us with $(1-p)\phi$ unpruned parameters. Let $f = 1-p$. We first calculate the memory required to store the compressed model states i.e. all model states except θ^{16}. For each of these tensors, we only need to maintain data for $f\phi$ parameters. This adds upto $18f\phi$ bytes ($2f\phi$ bytes for $\nabla\theta^{16}$, $4f\phi$ each for θ^{32} and $\nabla\theta^{32}$, and another $8f\phi$ for os). We also maintain a non-zero index per unpruned parameter. In our storage scheme, each non-zero index is a 32-bit integer. This requires another $4f\phi$ bytes. Storing the uncompressed θ^{16} state tensor adds a further 2ϕ bytes. Note that our optimizer step creates a temporary compressed copy of the half precision parameters at the end of the optimizer step (See Section 4.1.3). This adds another $2f\phi$ bytes. Adding everything together, the total memory consumption of model state storage in

bytes is :

$$M^{SAMO} = 18f\phi + 4f\phi + 2\phi + 2f\phi \tag{4.1}$$

$$= 24f\phi + 2\phi \tag{4.2}$$

$$= 24(1-p)\phi + 2\phi \tag{4.3}$$

$$= 20\phi - (24p - 6)\phi \tag{4.4}$$

$$= M^{default} - (24p - 6)\phi \tag{4.5}$$

In other words, the absolute amount of memory savings that SAMO provides is $(24p - 6)\phi$ bytes, where p is the fraction of parameters that have been pruned and ϕ is the total number of parameters before pruning. In Figure 4.2, we plot the percentage memory saved by SAMO as compared to default mixed-precision training. We observe that, SAMO requires a minimum sparsity of 0.25 to break even in terms of memory consumption. However, given that most DL pruning algorithms can comfortably prune 80-90% of the parameters, this is not an issue. In this range of sparsities, we observe that our method saves a significant 66-78% of memory required to store model states!

4.2 Exploiting SAMO for Improving Parallel Training Performance

When computing on a single GPU, SAMO simply reduces memory consumption with some overheads in the backward pass (compression of gradients) and the optimizer step (expansion of parameters). Hence, when training on a single GPU, SAMO does not lead to any performance improvements. This is because the total number of floating point operations in the forward and

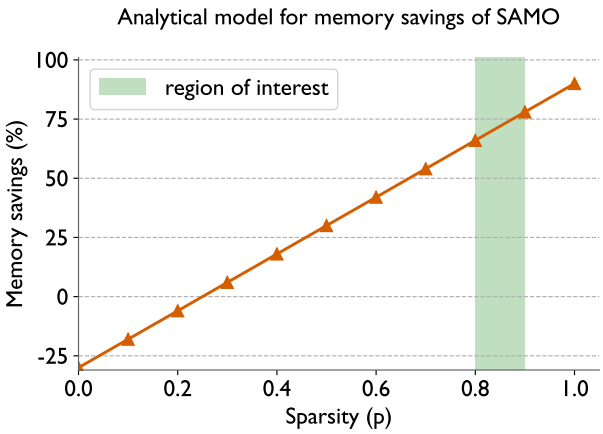

Figure 4.2: Percentage memory saved by SAMO as compared to default mixed-precision training. Sparsity here refers to the proportion of parameters that have been pruned. SAMO can save around 66-78% memory in a range of 0.8-0.9 sparsity, which is typical for most pruning algorithms in deep learning.

backward pass is unchanged (since we still compute in dense). In this section, we discuss how parameter pruning and SAMO can be used to optimize the performance of multi-GPU training.

The main performance bottleneck in parallel neural network training is communication. GPUs perform computation on data at a much faster rate than that of data communication between them on modern HPC interconnects. This problem is only exacerbated when training larger models, which require a correspondingly larger number of GPUs on a cluster. Thus, designing algorithms that can decrease the amount of communication can greatly benefit parallel deep learning. We now discuss how the application of SAMO on a pruned neural network can reduce communication in parallel training. We use AxoNN [56] (discussed in Chapter 3), which implements a hybrid of inter-layer parallelism (point-to-point communication) and data parallelism (collective communication), to demonstrate the efficacy of our optimizations.

4.2.1 Optimizing collective communication in data parallelism

First, let us see how our optimizations can decrease the overhead of collective communication in the data parallel phase. After the end of the forward and backward pass, AxoNN synchronizes the local gradients of each GPU via an all-reduce. In Section 4.1.1, we showed how SAMO stores the 16-bit gradients in a compressed format i.e. only for the unpruned parameters. This allows us to reduce the size of collective communication messages by directly invoking AxoNN's all-reduce calls on the compressed tensor. This leads to a significant reduction in the collective communication time.

4.2.2 Optimizing point-to-point communication in inter-layer parallelism

AxoNN implements a hybrid of inter-layer and data parallelism by dividing the work among $G_{\text{inter}} \times G_{\text{data}}$ GPUs. When SAMO is used to reduce the memory required for training a neural network, we can reduce the number of GPUs required to deploy a single instance of the neural network i.e. decrease G_{inter}. This can allow us to use more GPUs for data parallelism, and increase G_{data}. A reduced G_{inter} has the effect of decreasing the time spent in point-to-point communication thereby increasing the efficiency of inter-layer parallelism. We now provide a proof for this claim. We use the following notations:

- B - Batch size
- mbs - The size of each microbatch
- G - Number of GPUs
- t_f - Time spent in computation on a microbatch of size mbs during the forward pass through

the entire model

- t_b - Time spent in computation on a microbatch of size mbs during the backward pass through the entire model

Note that t_f and t_b do not take the point-to-point communication cost into account. They just denote the compute time for the forward and backward pass across all the layers.

The time spent in point-to-point communication can be divided into two parts: the bubble time and the transmission time. A GPU experiences a pipeline bubble when there aren't enough microbatches in the pipeline to keep all of the GPUs busy. As shown in Figure 4.3, different GPUs experience pipeline bubbles at different points in time. But a common theme is that pipeline bubbles occur towards the beginning and end of the computation of a batch. We define the transmission time as the total time spent in sending messages in the pipeline.

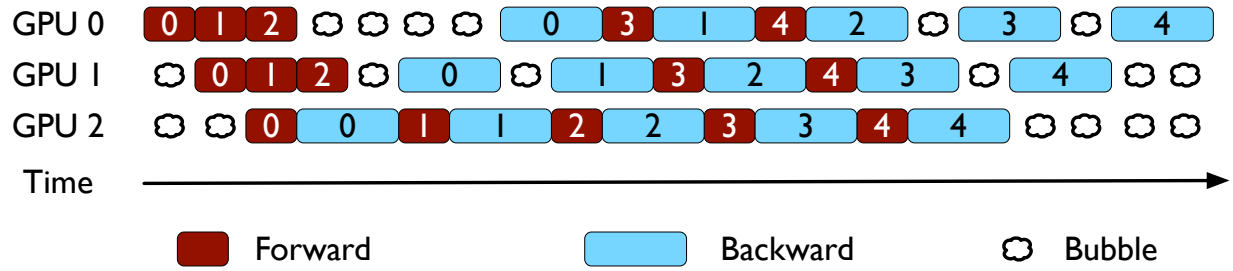

Figure 4.3: Illustration of how a batch is computed in inter-layer parallelism on three GPUs ($G_{\text{inter}} = 3$). In this example, we have divided the input batch into 5 microbatches (numbered 0 to 4). The red and blue colors denote forward and backward passes of microbatches respectively. We assume that the forward pass takes one unit of time and the backward pass takes two units of time. We observe that on each GPU, the pipeline bubble time accounts for 6 units, which equals the time to do $G_{\text{inter}} - 1 = 2$ forward passes and $G_{\text{inter}} - 1 = 2$ backward passes.

Let t_{bubble} and t_{send} denote the bubble time and transmission time respectively. Narayanan et al. [42] show that t_{bubble} equals the time it takes to complete forward and backward passes for $G_{\text{inter}} - 1$ microbatches on any GPU. We can also see this in Figure 4.3, wherein we observe

that the bubble time for a pipeline with $G_{\text{inter}} = 3$ equals the time to do two forward and two backward passes. Assuming uniform distribution of compute, the time to complete the forward and backward pass of a microbatch on a single GPU is $\frac{t_f + t_b}{G_{\text{inter}}}$.

Thus, the bubble time can be calculated as,

$$t_{\text{bubble}} = (G_{\text{inter}} - 1) \times (\frac{t_f + t_b}{G_{\text{inter}}}) \tag{4.6}$$

$$= (t_f + t_b) \times (1 - \frac{1}{G_{\text{inter}}}) \tag{4.7}$$

Now, taking the derivative of t_{bubble} with G_{inter}, we can show that the pipeline bubble time is a monotonically increasing function of G_{inter}:

$$\frac{\partial t_{\text{bubble}}}{\partial G_{\text{inter}}} = \frac{t_f + t_b}{G_{\text{inter}}^2} > 0 \tag{4.8}$$

Since SAMO can help in decreasing G_{inter} via its memory savings, we can conclude that it can be used to optimize the pipeline bubble time. Note that in Equation 4.8, we observe that the gradient w.r.t. G_{inter} is inversely proportional to its square. Thus, with a progressive increase in model size (which entails a corresponding increase in G_{inter}), we expect diminishing returns in the bubble time improvement.

The transmission time t_{send} is proportional to the number of messages sent and received by each GPU. Each GPU sends and receives four messages per microbatch, two each in the forward and backward passes. Let us now derive the total number of microbatches each GPU computes on. First, AxoNN divides the input batch into G_{data} shards, one for each inter-layer parallel group. Next, each inter-layer parallel group breaks this batch shard into microbatches of size

mbs. These microbatches are processed by every GPU in the inter-layer parallel group. Thus the total number of microbatches computed upon by every GPU is $\frac{B}{G_{\text{data}} \times \mathrm{mbs}}$. Thus, we can express t_{send} as,

$$t_{\text{send}} \propto 4 \times \frac{B}{\mathrm{mbs} \times G_{\text{data}}} \tag{4.9}$$

$$\propto 4 \times \frac{B}{\mathrm{mbs}} \times \frac{G_{\text{inter}}}{G} (\because G_{\text{inter}} \times G_{\text{data}} = G) \tag{4.10}$$

Taking the derivative of Equation 4.10 w.r.t. G_{inter} shows that t_{send} is a monotonically increasing function of G_{inter}:

$$\frac{\partial t_{\text{send}}}{\partial G_{\text{inter}}} \propto \frac{B}{\mathrm{mbs} \times G} > 0 \tag{4.11}$$

Hence, we can see that using SAMO to decrease G_{inter} can also help us decrease the transmission time for point-to-point communication in inter-layer parallelism. Thus, we have shown how the memory optimizations in SAMO can be exploited to reduce the collective communication pertaining to data parallelism and point-to-point communication pertaining to inter-layer parallelism respectively. Later, in Section 4.3, we provide performance profiles that demonstrate reduction in communication times as empirical evidence for the claims we have made in this section.

4.3 Results

Table 4.1 lists the neural networks used in this study.

Table 4.1: List of neural networks used in this study. For each model, we list the minimum and maximum number of GPUs used in our strong scaling runs. We choose the minimum and maximum GPU counts such that the ratio of batch size to number of GPUs is 4 and 1 respectively.

Neural Network	# Parameters	Batch Size	No. of GPUs
WideResnet-101 [81]	126.89M	128	16–128
VGG-19 [82]	143.67M	128	16–128
GPT-3 XL [1]	1.3B	512	64–512
GPT-3 2.7B [1]	2.7B	512	64–512
GPT-3 6.7B [1]	6.7B	1024	128–1024
GPT-3 13B [1]	13B	2048	256–2048

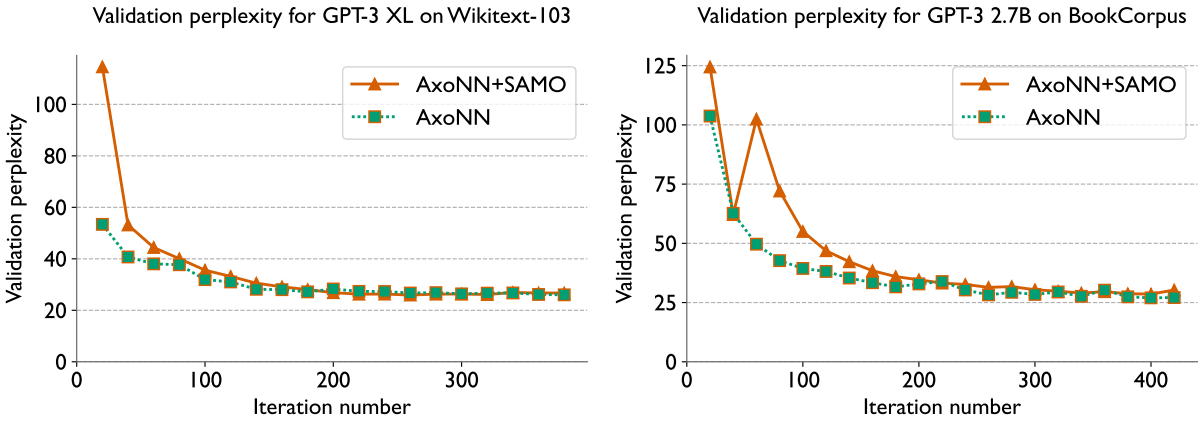

Figure 4.4: Validation perplexities for GPT-3 XL (left) and GPT-3 2.7B (right) on 64 and 128 GPUs of Summit respectively. For AxoNN +SAMO, we prune both models to a sparsity of 90% using [4]. We use the same hyperparameters as Brown et al. [1] and train on the Wikitext-103 [5] and BookCorpus datasets [6].

4.3.1 Statistical efficiency

We verify the statistical efficiency of AxoNN +SAMO by training GPT-3 XL [1] and GPT-3 2.7B [1] to completion at a sparsity of 90%. We use You et al.'s algorithm [4] to prune a neural network for AxoNN +SAMO. Figure 7.7 illustrates the results of this experiment. We observe that (1) the final validation perplexities for the pruned networks trained with AxoNN +SAMO match those of the unpruned network trained with AxoNN and (2) both AxoNN and AxoNN +SAMO reach the final validation perplexities in similar number of iterations. This verifies the

correctness of our implementation.

4.3.2 Strong scaling performance

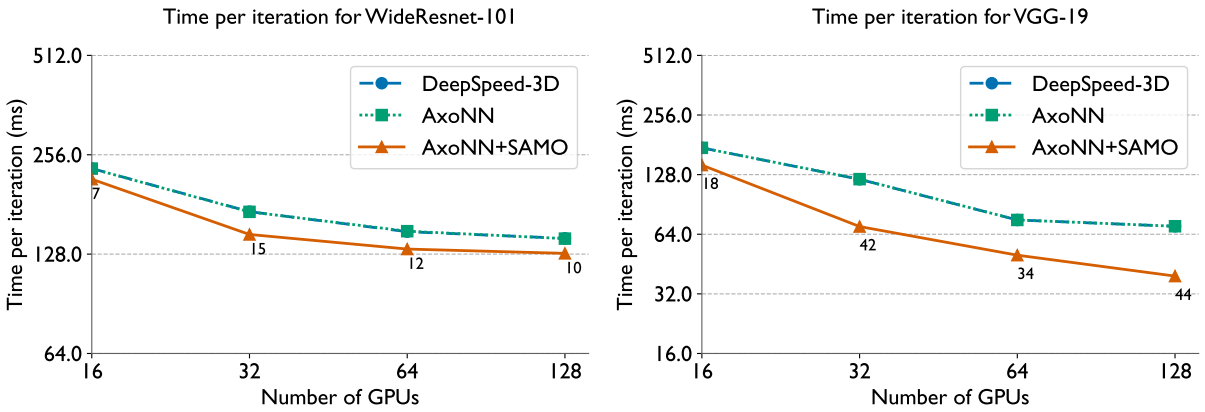

Figure 4.5: Time per iteration (batch time) for a strong scaling study of WideResnet-101 (left) and VGG-19 (right) on Summit. We prune the models to a sparsity of 90% for AxoNN +SAMO (see Table 4.1 for batch sizes). We annotate AxoNN +SAMO's line with its percentage speedup over AxoNN.

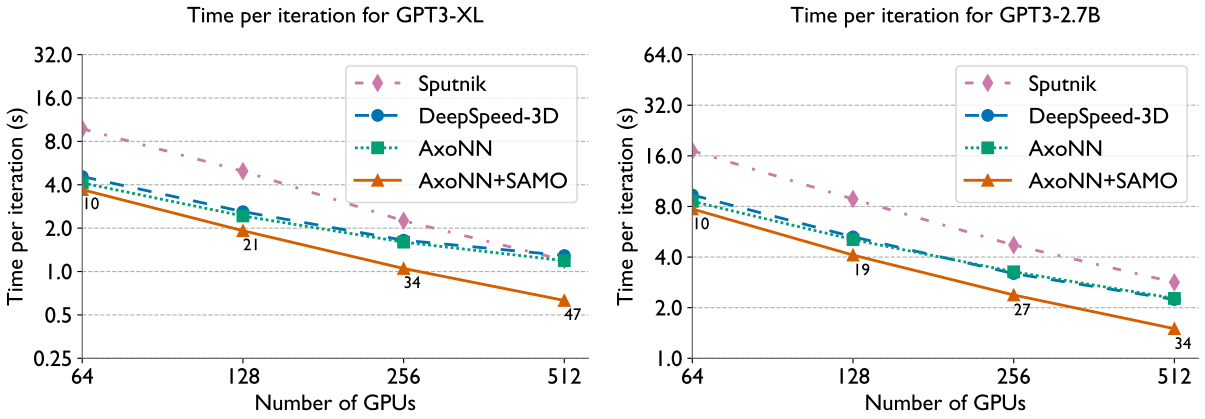

Figure 4.6: Time per iteration (batch time) for a strong scaling study of GPT-3 XL (left) and GPT-3 2.7B (right) on Summit. We prune the models to a sparsity of 90% for AxoNN +SAMO and Sputnik (see Table 4.1 for batch sizes). We annotate AxoNN +SAMO's line with its percentage speedup over AxoNN.

Next, we illustrate the results of our strong scaling experiments on WideResnet-101 and VGG-19 in Figure 4.5, GPT-3 XL and GPT-3 2.7B in Figure 4.6, and on GPT-3 6.7B and GPT-3

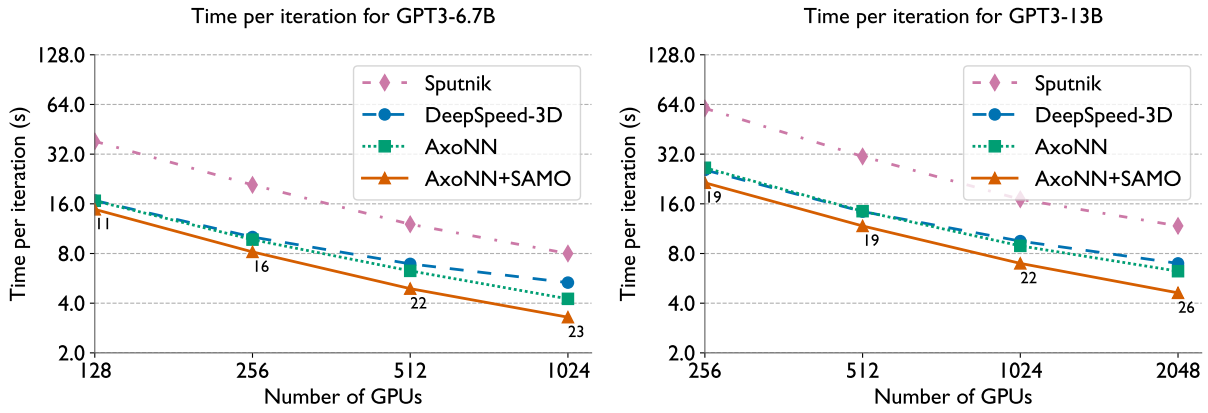

Figure 4.7: Time per iteration (batch time) for a strong scaling study of GPT-3 6.7B (left) and GPT-3 13B (right) on Summit. We prune the models to a sparsity of 90% for AxoNN +SAMO and Sputnik (see Table 4.1 for batch sizes). We annotate AxoNN +SAMO's line with its percentage speedup over AxoNN.

13B in Figure 4.7. The CNNs used in this study are nearly 10–100× smaller than the GPT-3 based models (see Table 4.1). Hence, all of AxoNN, DeepSpeed-3D and AxoNN +SAMO are able to run these architectures in a pure data parallel configuration, with a full copy of the network on each GPU. Thus the only communication here is the all-reduce on the network gradients. We illustrate these results in Figure 4.5. We observe similar batch times for both AxoNN and DeepSpeed-3D. This is explained by the fact that both these frameworks have very similar NCCL-based implementations of data parallelism. Our approach yields speedups of 7–12% over WideResnet-101 and 18–44% over VGG-19. While both these speedups are significant, SAMO seems to benefit the latter architecture more than the former. This is because the WideResnet-101 architecture spends nearly 1.5× more time in the computation phase as compared to VGG-19. Also, both these models have similar number of parameters and thus similar communication costs in the data-parallel all-reduce. Thus the proportion of the batch time spent by the WideResnet-101 architecture in communication is significantly smaller than VGG-19. Since our approach optimizes communication, the benefits for WideResnet-101 are smaller than that of VGG-19. Note

that we do not run Sputnik for the CNNs as the library does not support sparse convolutions.

Let us now discuss the much larger GPT-3 based neural networks. These networks are too large to fit on a single GPU and are thus trained using hybrid parallelism. First, we observe that the performance of the sparse matrix computation library, Sputnik is significantly worse than both of our dense baselines – AxoNN and DeepSpeed-3D, as well as AxoNN +SAMO (Figures 4.6 and 4.7). This is in spite of the fact that the number of floating point operations computed by Sputnik is 10% of the other methods. This is in agreement with our observations in Figure 4.1 for fully connected layers on a single GPU. Thus, AxoNN +SAMO ends up being nearly twice as fast as Sputnik across all the GPT-3 style neural networks. It is evident from Figures 4.6 and 4.7 that augmenting AxoNN with our optimizations significantly improves its performance at scale. Our method speeds up the training of GPT-3 XL by 10–47%, GPT-3 2.7B by 10–24%, GPT-3 6.7B by 11–23% and GPT-3 13B by 19–26%. The speedups over DeepSpeed-3D are larger – 19–51%, 17–33%, 12–38% and 16.4–34% respectively for the four models.

We also present the percentage of peak half precision throughputs obtained for GPT-3 13B in Table 4.2. We observe a significant reduction in the GPU utilization with increasing GPU counts for DeepSpeed-3D and AxoNN. This is a consequence of increasing communication to computation ratios. For both frameworks, the peak half precision throughput drops to around 20% at the largest profiled GPU counts. However, with AxoNN +SAMO, we observe a smaller reduction in hardware utilization, with a peak throughput of around 30% for the largest GPU count. This serves as empirical evidence of the fact that our optimizations indeed decrease the amount of communication in parallel training.

Since our optimizations are geared toward reducing the communication costs of training, we expect larger improvements over AxoNN as the number of GPUs increase. Again, this is

Table 4.2: Percentage of peak half precision throughput for a strong scaling study of GPT-3 13B on Summit (see Table 4.1 for batch sizes). We prune the models to a sparsity of 90% for AxoNN +SAMO and Sputnik.

# GPUs	Sputnik	DeepSpeed-3D	AxoNN	AxoNN+SAMO
256	18.9	44.6	43.3	**53.4**
512	18.5	39.9	39.7	**48.8**
1024	16.8	30.1	32.2	**41.1**
2048	12.2	20.6	22.9	**31.0**

because a larger proportion of time is spent in communication as we increase the scale of training. We find our observations in Figures 4.6 and 4.7 to be in agreement with this hypothesis. We indeed observe the largest speedups for the largest GPU counts, which are 47% and 34% for GPT-3 XL and GPT-3 2.7B on 512 GPUs, 23% for GPT-3 6.7B on 1024 GPUs, and 26% for GPT-3 13B on 2048 GPUs.

4.3.3 Performance Breakdowns

To verify that the speedups over AxoNN are indeed due to reduction in communication times, we profile the GPT-3 2.7B model on 128, 256 and 512 GPUs and provide breakdowns of the batch times in Figure 4.8. We divide the batch time into its non-overlapping phases, namely the compute (forward and backward pass), point-to-point communication, pipeline bubble (due to inter-layer parallelism), and collective communication (due to data parallelism). We use the CUDA Event API to profile the time spent in each of these phases.

At 128 GPUs, we observe that training is dominated by the point-to-point communication time. However as the number of GPUs increase, the proportion of time spent in the point-to-point communication also decreases. Note that this is in line with Equation 4.10, wherein we showed that the messaging time is inversely proportional to the number of GPUs.

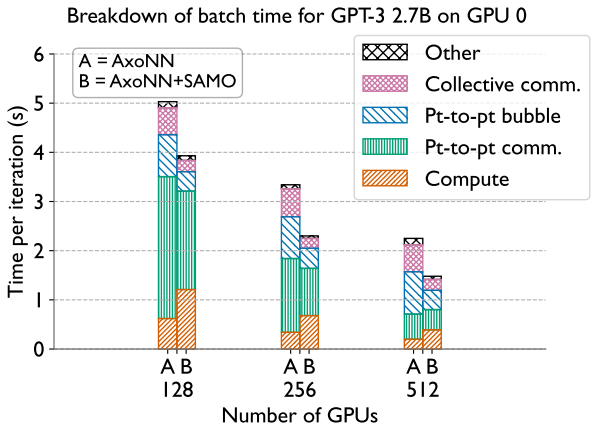

Figure 4.8: Breakdown of batch time for GPT-3 2.7B on Summit. We divide the batch time into its non-overlapping phases – computation, point-to-point communication, pipeline bubble (due to inter-layer parallelism), and collective communication (due to data parallelism). We use the CUDA Event API to profile the cumulative time spent in each of these phases.

We observe that the primary reason for AxoNN +SAMO's improvement over AxoNN on 128 GPUs is due to a speedup in the point-to-point communication times. The absolute reduction in this time is 18% of AxoNN's batch time. The improvements in the collective and pipeline bubble times account for 6% and 9% of AxoNN's batch time. Thus for smaller GPU counts, we conclude that AxoNN +SAMO provides speedups primarily because of the improvements in the point-to-point communication times. The difference in the compute times is the overhead of compressing the parameter gradients at every backward pass (See Section 4.1.3). The overhead accounts for 12% of AxoNN's batch time and is significantly overcome by the 33% (18+6+9) improvement in the total communication time. We think that these overheads can be reduced by kernel level optimizations such as fusing the compression operation with the backward pass kernels. However, this is out of the scope of our current work.

At 256 GPUs, the point-to-point communication time is still dominant but not as much as 128 GPUs. In this case, the improvement in the point-to-point communication time accounts for a 16.17% of AxoNN's batch time. As compared to 128 GPUs, the improvements in the bubble and

collective communication times account for a significantly larger proportion of AxoNN's batch time - 13.17% and 11.08%. The overhead in this case is 10.18% of the total batch time.

At 512 GPUs, we notice a very minor reduction in the point-to-point communication time. The reduction in the bubble and collective communication time account for 15% and 21% of AxoNN's batch time respectively. The reduction in the point-to-point communication only improves the batch times by 4%. In this case, the overhead of compressing gradients is 8% of AxoNN's batch time, which is again overcome by 40% (15+21+4) improvement in the total communication times.

Chapter 5: A 4D Hybrid Algorithm to Scale Parallel Training to Thousands of GPUs

This chapter is based on prior work [83], which introduces a four-dimensional (4D) approach to optimize communication in parallel training. This 4D approach is a hybrid of 2D tensor, FSDP [23, 25] and data parallelism. They key innovation of this work is an analytical communication performance model, which identifies high-performing configurations within the large search space defined by our 4D algorithm. Additionally, it introduces strategies to aggressively overlap communcation with computation and BLAS kernel tuning to improve compute performance. All of this leads to efficient scaling of training to tens of thousands of GPUs, acheiving peak flop/s (bf16) of 620.1 Petaflop/s on Perlmutter, 1.381 Exaflop/s on Frontier and 1.423 Exaflop/s on Alps.

5.1 Innovations Realized

Training deep neural networks on a single GPU involves processing subsets of the data called batches through the layers of a DNN in the forward pass to compute a loss, computing the gradient of the loss in a backward pass via backpropagation, and updating the parameters (also called "weights") in the optimizer step. These three steps are repeated iteratively until all batches have been consumed, and this entire training process is referred to as an epoch. We now describe

our novel approach to scaling the computation in the steps described above in the context of large multi-billion parameter neural networks on thousands of GPUs.

5.1.1 A Four-Dimensional Hybrid Parallel Approach

We have designed a hybrid parallel approach that combines data parallelism with three-dimensional (3D) parallelization of the matrix multiplication routines.

Data Parallelism: In order to use a hybrid approach that combines data with tensor parallelism, we organize the total number of GPUs, G, into a virtual 2D grid, $G_{\text{data}} \times G_{\text{tensor}}$. This results in G_{data} groups of G_{tensor} GPUs each. We use data parallelism across the G_{data} groups, and tensor parallelism within each group. Each G_{data} group collectively has a full copy of the neural network and is tasked to process a unique shard of the input batch. At the end of an input batch, all groups have to synchronize their weights by issuing all-reduces on their gradients after every batch (this is also referred to as an iteration).

3D Parallel Matrix Multiplication (3D PMM): Next, we use each GPU group, composed of G_{tensor} GPUs to parallelize the work within their copy of the neural network. This requires distributing the matrices, and parallelizing the computation within every layer of the neural network across several GPUs. Note that most of the computation in transformers is comprised of large matrix multiplications within fully connected (FC) layers. Hence, in this section, we will focus on parallelizing FC layers with a 3D PMM algorithm.

We now describe how a single layer is parallelized, and the same method is applied to all the layers in the neural network. Each FC layer computes one half-precision (fp16 or bf16) matrix multiplication (input activation, I multiplied by the layer's weight matrix, W) in the forward

pass and two half-precision matrix multiplications (MMs) in the backward pass ($\frac{\partial L}{\partial O} \times W^\top$ and $I^\top \times \frac{\partial L}{\partial O}$, where L is the training loss, and O is the output activation.) Thus, parallelizing an FC layer requires parallelizing these three MM operations across multiple GPUs.

We adapt Agarwal et al.'s 3D parallel matrix multiplication algorithm [7], for parallelizing our MMs. The 3D refers to organizing the workers (GPUs) in a three-dimensional virtual grid. So, we organize the G_{tensor} GPUs further into a virtual 3D grid of dimensions $G_x \times G_y \times G_z$ (Figure 5.1). We do 2D decompositions of both I and W into sub-blocks and map them to orthogonal planes of the 3D grid. In the figure below, I is distributed in the XZ plane, and copied in the Y dimension. W is distributed in the XY plane and copied along the Z dimension. Once each GPU has a unique sub-block of I and W, it can compute a portion of the O matrix, which can be aggregated across GPUs in the X direction using all-reduces.

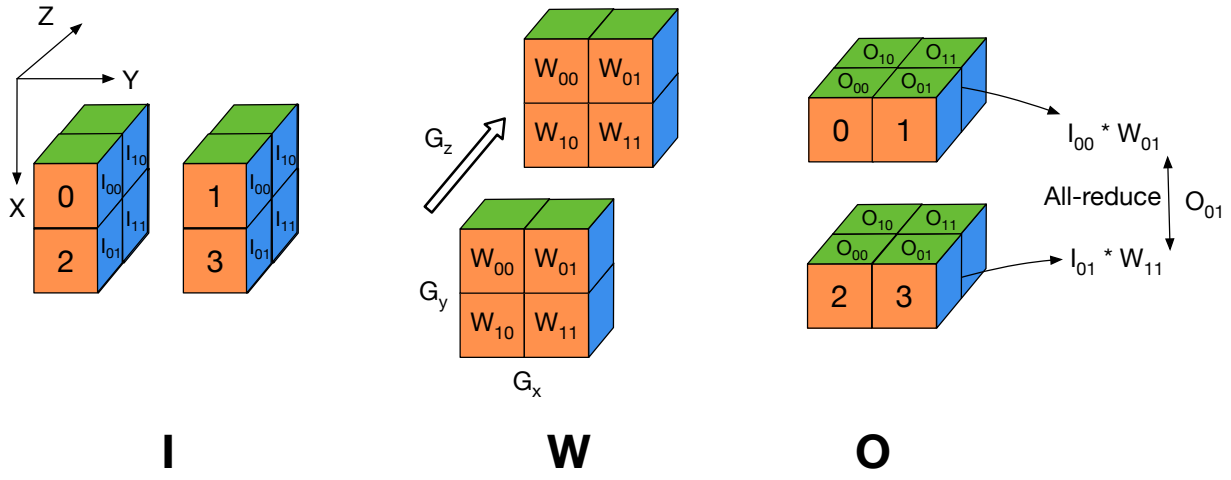

Figure 5.1: Parallelization of a matrix multiply in an FC layer with Agarwal's 3D parallel matrix multiplication algorithm [7] on eight GPUs organized in a $2 \times 2 \times 2$ topology. We use G_x, G_y, and G_z to refer to the number of GPUs along the three dimensions of the virtual grid topology.

We modify Agarwal's algorithm to reduce memory consumption, and instead of making copies of W along the Z-axis, we further shard W along the Z-axis and denote these sub-shards as $\hat{W}$. Algorithm 3 presents the forward and backward pass operations on GPU $g_{i,j,k}$, and we can

observe that the sharding of W results in all-gather operations before the local matrix multiplication on each GPU can proceed.

Algorithm 3 Tensor parallel algorithm for $g_{i,j,k}$ in a $G_x \times G_y \times G_z$ grid. Communication operations highlighted in blue.

1: **function** TENSOR_PARALLEL_FORWARD_PASS($I_{k,j}$, $\hat{W}_{j,i}$)
2: $W_{j,i}$ = ALL-GATHER$_z$($\hat{W}_{j,i}$)
3: $\hat{O}_{k,i} = I_{k,j} \times W_{j,i}$
4: $O_{k,i} \leftarrow$ ALL-REDUCE$_y$($\hat{O}_{k,i}$)
5: // Cache $I_{k,j}$ and $W_{j,i}$ for the backward pass
6: **return** $O_{k,i}$
7: **end function**
8:
9: **function** TENSOR_PARALLEL_BACKWARD_PASS($\frac{\partial L}{\partial O_{k,i}}$)
10: Retrieve $I_{k,j}$ and $W_{j,i}$ from cache
11: $\frac{\partial \hat{L}}{\partial I_{k,j}} \leftarrow \frac{\partial L}{\partial O_{k,i}} \times W_{j,i}^{\top}$
12: $\frac{\partial L}{\partial I_{k,j}} \leftarrow$ ALL-REDUCE$_x$($\frac{\partial \hat{L}}{\partial I_{k,j}}$)
13: $\frac{\partial \hat{L}}{\partial \hat{W}_{j,i}} \leftarrow I_{k,j}^{\top} \times \frac{\partial L}{\partial O_{k,i}}$
14: $\frac{\partial L}{\partial \hat{W}_{j,i}} \leftarrow$ REDUCE-SCATTER$_z$($\frac{\partial \hat{L}}{\partial \hat{W}_{j,i}}$)
15: **return** $\frac{\partial L}{\partial I_{k,j}}$, $\frac{\partial L}{\partial \hat{W}_{j,i}}$
16: **end function**

In the forward pass, after the local (to each GPU) matrix-multiply on line 3, we do an all-reduce to aggregate the output activations (line 4). In the backward pass, there are two matrix multiplies on lines 11 and 13, and corresponding all-reduce and reduce-scatter operations in lines 12 and 14 to get the data to the right GPUs.

Parallelizing an entire network: The approach of parallelizing a single layer in a deep neural network can be applied to all the layers individually. Let us consider a 2-layer neural network. If we use Algorithm 3 to parallelize each layer, the output O of the first layer would be the input to the other. However, notice in Figure 5.1 that O is distributed across the 3D virtual grid differently

than the input I. So to ensure that the second layer can work with O, we would need to transpose its weight matrix – essentially dividing its rows across the X-axis and columns across the Y-axis. This transpose needs to be done once at the beginning of training. Hence, to parallelize a multi-layer neural network, we simply 'transpose' the weights of every other layer by swapping the roles of the X- and Y- tensor parallel groups.

Note that the 4D algorithm (data + 3D PMM) discussed in this section is a generalization of various state-of-the-art parallel deep learning algorithms. For example, if one were to employ only the Z axis of our PMM algorithm to parallelize training, it would reduce to Fully Sharded Data Parallelism (FSDP) [25] and ZeRO [23]. Similarly, if we employ the Z axis of 3D PMM and data parallelism simultaneously, then our algorithm reduces to Hybrid Sharded Data Parallelism [25] and ZeRO++ [84]. If we use the X axis of our 3D PMM algorithm along with the 'transpose' scheme discussed in the previous paragraph, our 4D algorithm reduces to Shoeybi et al.'s Megatron-LM [39]. Finally, when all four dimensions of our 4D algorithm are being used, this is similar to a hybrid scheme that combines data parallelism, FSDP, and two-dimensional tensor parallelism.

5.1.2 A Performance Model for Identifying Near-optimal Configurations

When assigned a job partition of G GPUs, we have to decide how to organize these GPUs into a 4D virtual grid, and how many GPUs to use for data parallelism versus the different dimensions of 3D parallel martix multiplication. To automate the process of identifying the best performing configurations, we have developed a performance model that predicts the communication time of a configuration based on the neural network architecture, training hyperparameters,

and network bandwidths. Using these predictions, we can create an ordered list of the best performing configurations as predicted by the model. We describe the inner-workings of this model below.

We primarily focus on modeling the performance of the collective operations in the code, namely all-reduces, reduce-scatters, and all-gathers. We first list the assumptions we make in our model:

- *Assumption-1:* The ring algorithm [60] is used for implementing the all-reduce, reduce-scatter, and all-gather collectives.
- *Assumption-2:* For collectives spanning more than one compute node, the ring is formed such that the number of messages crossing node boundaries is minimized.
- *Assumption-3:* The message sizes are large enough, and hence, message startup overheads can be ignored. In other words, if a process is sending a message of n bytes, then we assumed that the transmission time is simply $\frac{n}{\beta}$, where β is the available bandwidth between the two processes.
- *Assumption-4:* We only model the communication times and ignore the effects of any computation taking place on the GPUs.
- *Assumption-5:* We assume the same peer-to-peer bidirectional bandwidth, β_{inter}, between every pair of nodes.

We use the analytical formulations in Thakur et al. [60] and Rabenseifner [85] for modeling the performance of ring algorithm based collectives. Let $t_{\text{AG},z}$ denote the time spent in the all-gather across the Z-tensor parallel groups (line 2 of Algorithm 3). Similarly, we use $t_{\text{RS},z}$, $t_{\text{AR},y}$

and $t_{\text{AR},x}$ to refer to the time spent in the collectives in lines 14, 4, and 12 respectively. Similarly, we use $t_{\text{AR,data}}$ for the time spent in the data parallel all-reduce. Then, we can model these times as follows,

$$t_{\text{AG},z} = \frac{1}{\beta} \times (G_z - 1) \times \frac{k \times n}{G_x \times G_y \times G_z} \tag{5.1}$$

$$t_{\text{RS},z} = \frac{1}{\beta} \times \left(\frac{G_z - 1}{\mathrm{G_z}} \right) \times \frac{k \times n}{G_x \times G_y} \tag{5.2}$$

$$t_{\text{AR},y} = \frac{2}{\beta} \times \left(\frac{G_y - 1}{G_y} \right) \times \frac{m \times n}{G_z \times \mathrm{G}_x} \tag{5.3}$$

$$t_{\text{AR},x} = \frac{2}{\beta} \times \left(\frac{G_x - 1}{G_x} \right) \times \frac{m \times k}{G_z \times G_y} \tag{5.4}$$

$$t_{\text{AR,data}} = \frac{2}{\beta} \times \left(\frac{G_{\text{data}} - 1}{G_{\text{data}}} \right) \times \frac{k \times n}{G_x \times G_y \times G_z} \tag{5.5}$$

The total communication time for a single layer, t_{comm} is simply the sum of Equations 5.1 through 5.5:

$$t_{\text{comm}} = t_{\text{AG},z} + t_{\text{RS},z} + t_{\text{AR},y} + t_{\text{AR},x} + t_{\text{AR,data}} \tag{5.6}$$

For layers with 'transposed' weight matrices as discussed at the end of Section 5.1.1, we need to swap the values of G_x and G_y. And finally, to model the communication time for the entire network, we apply Equation 5.6 to all of its layers, and take a sum of the times.

In the equations derived above, we made a simplifying assumption that all collectives in our hybrid parallel method can achieve the highest peer-to-peer bandwidth, denoted by β. However, since several collectives are often in operation at once, the actual bandwidth achieved for a collective operation among a group of GPUs depends on the placement of processes in our 4D virtual grid to the underlying hardware topology (nodes and network) [86–89]. For example, process groups that are contained entirely within a node can experience higher bandwidths than those containing GPUs on different nodes. Next, we model the specific bandwidths used in Equations 5.1 through 5.5.

To model the process group bandwidths, we begin by assuming a hierarchical organization of process groups: X-tensor parallelism (innermost), followed by Y-tensor parallelism, Z-tensor parallelism, and data parallelism (outermost). As a concrete example, if we have eight GPUs, and set $G_x = G_y = G_z = G_{\text{data}} = 2$, then the X-tensor parallel groups comprise of GPU pairs (0,1), (2,3), (4,5), and (6,7). Similarly, the Y-tensor parallel groups would comprise of GPU pairs (0,2), (1,3), (4,6), and (5,7), and so on.

Now let $\vec{G} = (G_x, G_y, G_z, G_{\text{data}})$ be the tuple of our configurable performance parameters, arranged in order of the assumed hierarchy. Let $\vec{\beta} = (\beta_x, \beta_y, \beta_z, \beta_{\text{data}})$ be the effective peer-to-peer bandwidths for collectives issued within these process groups. We use $\vec{\beta}_i$ and $\vec{G}_i$ to represent the i^{th} elements of these tuples ($0 \leq i \leq 3$). Also, let G_{node} refer to the number of GPUs per node. Now let us model each β_i i.e. the bandwidth available to the GPUs in the process groups at the i^{th} level of the hierarchy.

Case 1: GPUs in the process group lie within a node – in our notation, this is the scenario when $\prod_{j=0}^{i} G_j \leq G_{\text{node}}$.

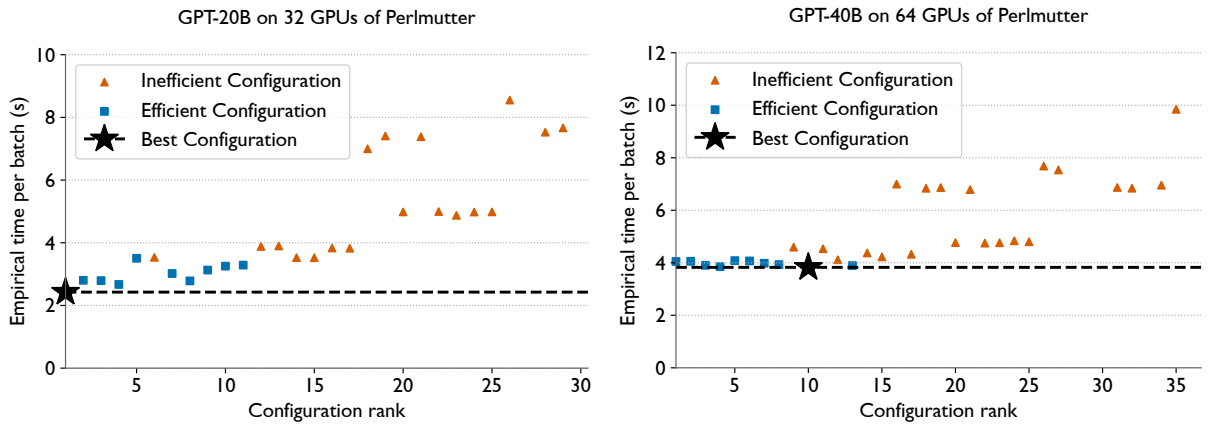

Figure 5.2: Plots validating the performance model by comparing the observed time per batch and the rank ordered by the model for two neural networks: GPT-20B (left) and GPT-40B (right).

The bandwidth $\vec{\beta}_i$ is determined by two primary factors: (i) the size of the ith process group, G_i, and (ii) the cumulative product of the sizes of all preceding process groups, $\prod_{j=0}^{i-1} G_j$. Given that the number of GPUs per node is typically small, the number of possible scenarios is also small. Therefore, we can profile the bandwidths for all potential configurations in advance and store this information in a database. Specifically, we generate all possible two-dimensional hierarchies of process groups (G_0, G_1) such that $G_0 \times G_1 \leq G_{\text{node}}$, and then perform simultaneous collectives within the outer process groups of size G_1 with a large message size of 1 GB. We record the achieved bandwidths for this tuple in our database. Then, for a given model, when we need the predicted bandwidths for the i^{th} process group, we retrieve the bandwidth recorded for the tuple $(G_0 = \prod_{j=0}^{i-1} G_j, G_1 = G_i)$.

Case 2: GPUs in the process group are on different nodes – in our notation, this is the scenario when $\prod_{j=0}^{i} G_j > G_{\text{node}}$.

For process groups spanning node boundaries, the approach of recording all possible configurations in a database is not feasible due to the vast number of potential scenarios, given the large number of possible sizes of these groups in a multi-GPU cluster. Therefore, we develop a

simple analytical model for this scenario, which predicts the achieved bandwidths as a function of the inter-node bandwidths (β_{inter}), process group sizes ($\vec{G}$), and the number of GPUs per node (G_{node}).

First, let's first explore two simple examples to build some intuition. In Figure 5.3 (left), we demonstrate a scenario with a single process group spanning eight GPUs on two nodes, with four GPUs on each node. In this case, the ring messages crossing node boundaries (i.e. the link between GPUs 1 and 4, and between GPUs 6 and 3) will be the communication bottleneck. Since we assumed β_{inter} to be the bidirectional bandwidth between node pairs, we can set $\beta_i = \beta_{\text{inter}}$.

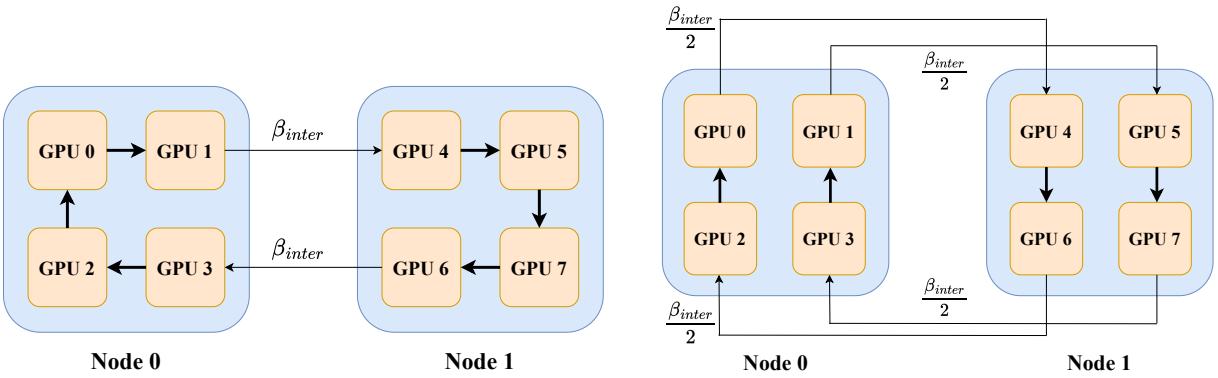

Figure 5.3: (Left) Formation of a single ring connecting eight GPUs across two nodes for collective communication operations such as all-reduce, reduce-scatter, or all-gather, with the model assigning the maximum available inter-node bandwidth to this ring. (Right) Creation of two independent rings, each spanning four GPUs across two nodes, where the model assumes the available inter-node bandwidth is evenly divided between the two rings.

Another possible scenario is when there are multiple simultaneous collectives taking place between two nodes. For example, consider Figure 5.3 (right), wherein GPUs $(0, 4, 6, 2)$ and GPUs $(1, 5, 7, 3)$ are executing two independent collectives using the ring algorithm simultaneously. In this case, the available inter-node bandwidth will be shared between these two collectives and $\beta_i = \frac{\beta_{\text{inter}}}{2}$.

The first scenario occurs in the case when the process groups preceding the i^{th} process

group in the hierarchy are of size one, i.e. $G_j = 1 \; \forall j < i$. Whereas the second scenario occurs in the case when at least one of these preceding process groups is of a size > 1. In that case, we get multiple ring messages crossing node boundaries and the bandwidth gets distributed between the rings. However, note that the maximum reduction in the bandwidth is bounded by the total number of GPUs on each node, as there can't be more inter-node ring links than GPUs on a node. Equation 5.7 models all the scenarios to obtain the observed bandwidth:

$$\vec{\beta_i} = \frac{\beta_{\text{inter}}}{\min\left(G_{\text{node}}, \prod_{j=0}^{i-1} G_j\right)} \tag{5.7}$$

We use this bandwidth term in Equations 5.1 through 5.5 of our model. We use the model to create an ordered list of configurations, and then we can pick the top few configurations for actual experiments.

Validating the Performance Model: To validate the model, we collect the batch times for all possible configurations of the 4D virtual grid when training GPT-20B on 32 GPUs and GPT-40B on 64 GPUs of Perlmutter. Using the observed batch times, we label the ten fastest configurations as 'efficient' and the rest as 'inefficient'. When creating the validation plots, we rank the configurations using the ordering provided by the performance model. Figure 5.2 shows the empirical batch times on the Y-axis and the rank output by the performance model on the X-axis. The fastest configurations should be in the lower left corner. We observe that nine out of the top ten configurations predicted by the performance model are indeed 'efficient' as per their observed batch times. This shows that the model is working very well in terms of identifying the fastest configurations.

5.1.3 Automated Tuning of BLAS Kernels

In deep neural networks, a significant portion of the computational workload is matrix multiplications kernels or "matmuls", particularly in transformer models. These matmuls can be performed in one of three main modes based on whether the operands are transposed: NN, NT, and TN. Prior research has highlighted that NT and TN kernels are often less optimized than NN kernels in most BLAS libraries [90]. In our experiments, we found this discrepancy to be more pronounced when running transformers with large hidden sizes on the AMD MI250X GPUs of Frontier. For example, in the GPT-320B model (described in Table 5.1), we observed that a matrix multiply defaulting to the TN mode in PyTorch achieved only 6% of the theoretical peak performance, whereas other matmuls reached 55% of the peak.

To address this issue, we implemented an automated tuning strategy in which, during the first batch, each matmul operation in the model is executed in all three modes (NN, NT, and TN) and timed. We then select the most efficient configuration for each operation, which is subsequently used for the remaining iterations. This tuning approach ensures that our deep learning framework, AxoNN, avoids the pitfalls of using suboptimal matmuls that could significantly degrade performance. For the aforementioned 320B model, our BLAS kernel tuning approach successfully switches the poorly performing TN matmul with a nearly 8$\times$ faster NN matmul, thereby reducing the total time spent in computation from 30.1 seconds to 13.19s! Note that for other models used in Table 5.1, the speedups attained via tuning are relatively modest (See Figure 5.5) (right).

5.1.4 Overlapping Asynchronous Collectives with Computation

We use non-blocking collectives implemented in NCCL and RCCL on NVIDIA and AMD platforms respectively. This enables us to aggressively overlap the collective operations in AxoNN with computation, which can minimize communication overheads.

Overlapping All-reduces with Computation (OAR): In this performance optimization, we overlap the all-reduce across the X-tensor parallel groups in the backward pass (Line 12 of Algorithm 3) with the computation in Line 13. Once this computation has completed, we wait on the asynchronous all-reduce. Note that for layers with 'transposed' weight matrices, this communication happens across the Y-tensor parallel groups.

Overlapping Reduce-scatters with Computation (ORS): Next we overlap the reduce-scatters in the backward pass (line 14 of algorithm 3). The outputs of this reduce-scatter are the gradients of the loss w.r.t. the weights. These outputs are not needed until the backward pass is completed on all the layers of the neural network and we are ready to start issuing the all-reduces in the data parallel phase. Exploiting this, we issue these reduce-scatters asynchronously and only wait on them once all layers have finished their backward pass. This allows us to overlap the reduce-scatter of one layer with the backward pass computations of the layers before it.

Overlapping All-gathers with Computation (OAG): Our next optimization overlaps the all-gather operations in the forward pass (line 2 of Algorithm 3) with computation. We observe that this all-gather operation does not depend on any intermediate outputs of the forward pass. Leveraging this, we preemptively enqueue the all-gather for the next layer while the computation for the current layer is ongoing. At the start of training, we generate a topological sort of

the neural network computation graph to determine the sequence for performing the all-gathers. Subsequently, we execute them preemptively in this order.

Figure 5.4 shows the performance improvements from the three successive collective overlap optimizations (OAR: Overlap of all-reduces, ORS: Overlap of reduce-scatters, and OAG: Overlap of all-gathers). The baseline here refers to the scenario with no communication overlap. We also show the breakdown of the total time per batch into computation and communication. As we can see, the times spent in computation do not change significantly, however, the time spent in non-overlapped communication reduces with successive optimizations, leading to an overall speedup. For the 80B model in the figure, we see a performance improvement of 18.69% over the baseline on 8,192 GCDs of Frontier.

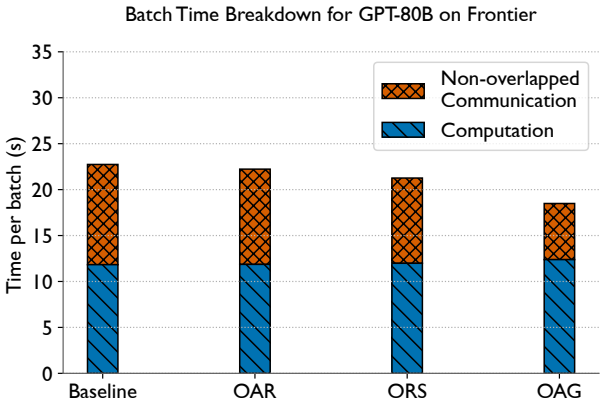

Figure 5.4: The impact of overlapping non-blocking collectives with computation on the training times of different sized models on 8,192 GCDs of Frontier.

5.2 How Performance Was Measured

All of our innovations are implemented in an open-source framework called AxoNN [56], which can be integrated easily as a backend in existing serial training codebases. This section provides details of the experimental setup for benchmarking training performance using AxoNN.

5.2.1 Applications: Model Architecture Details

We evaluate the effectiveness of our implementation by conducting experiments on a well-known neural network architecture: Generative Pre-trained Transformer (GPT) [1]. The GPT architecture is a popular transformer architecture [8] that has been used to train several large language models [1,40,41,91]. Table 5.1 presents the sizes of the different model architectures used in the experiments, and their important hyperparameters. Due to the extremely large activation memory requirements of training GPT models, we turn on activation checkpointing [77]. Additionally, we employ mixed precision (bf16/fp32) for all our training runs. We use bf16 since it has been shown to achieve the same performance and stability as fp32 [92], and it maintains the same range as fp32. This makes it a suitable choice over fp16, which has been known to be numerically unstable for LLM training.

Table 5.1: Architectural details of the GPT-style transformers [1] used in the performance experiments.

Model	# Parameters	# Layers	Hidden-Size	# Heads
GPT-5B	5B	24	4096	32
GPT-10B	10B	32	5120	40
GPT-20B	20B	32	7168	56
GPT-40B	40B	38	9216	72
GPT-60B	60B	56	9216	72
GPT-80B	80B	42	12288	96
GPT-160B	160B	84	12288	96
GPT-320B	320B	96	16384	128
GPT-640B	640B	192	16384	128

On Perlmutter, we use the sequential model training code from the Megatron-LM codebase [42], and parallelize it using AxoNN. However, on Frontier, we observed training instabilities with Megatron-LM, and switched to using LitGPT [93] for the model architectures on

Frontier and Alps. We parallelized LitGPT also using our 4D implementation in AxoNN. We conduct weak scaling experiments with the GPT-3 models, ranging from 5 billion to 320 billion parameters. We also conduct strong scaling experiments on Frontier using the 80 billion and 640 billion parameter models to predict the time-to-solution for 2 trillion tokens.

5.2.2 Systems and Environments

Our experiments were conducted on three supercomputers, Perlmutter at NERSC/LBL, Frontier at OLCF/ORNL, and Alps at CSCS. Each node on Perlmutter is equipped with four NVIDIA A100 GPUs, each with a DRAM capacity of 40 GB. On Frontier, each node has four AMD Instinct MI250X GPUs each with a DRAM capacity of 128 GB. Each MI250X GPU is partitioned into two Graphic Compute Dies (GCDs) and each 64 GB GCD can be managed independently by a process. On Alps, each node has four GH200 Superchips, where each H100 GPU has a DRAM capacity of 96 GB. Nodes on all systems have four HPE Slingshot 11 NICs, with each NIC capable of bidirectional link speeds of 25 GB/s.

In our Perlmutter experiments, we use CUDA 11.7, NCCL 2.15.5, and PyTorch 1.13. On Frontier, we use PyTorch 2.2.1 with ROCm 5.7 and RCCL 2.18.6. On Alps, we use PyTorch 2.4.0 with CUDA 12.5.1 and NCCL 2.22.3. On all the systems, we use the AWS OFI plugin (NCCL or RCCL) which enables us to use libfabric as the network provider on the Slingshot network, and provides high inter-node bandwidth. We want to note here that several runs on Perlmutter and Alps were done in a system-wide reservation, and even so, we noticed significant run-to-run performance variability. This was most likely due to network congestion [94] or file-system degradation [95] impacting performance.

5.2.3 Evaluation Metrics

In all our experiments, we run the training loop for ten iterations (batches), and report the average time per iteration (batch) for the last eight iterations to account for any performance variability due to initial warmup. We calculate half precision flop/s (often called "model flops") using Narayanan et al.'s analytical formulation [42] for the number of floating point operations in a transformer model. We did a small experiment to verify that this formulation matches the total number of floating point operations measured by Nsight Compute, an empirical tool. We compare this number against the theoretical (vendor advertised) peak performance of each GPU (312 Tflop/s per GPU on Perlmutter, 191.5 Tflop/s per GCD on Frontier, 989 Tflop/s per GPU on Alps), and report the achieved percentage of peak as well as the total sustained bf16 flop/s.

Since the vendor advertised peak performance is often not practically achievable, we also ran a simple GEMM benchmark on 1 GPU/GCD of Perlmutter/Frontier to gather empirically observed peak flop/s. We invoked equivalent cuBLAS and rocBLAS kernel calls to multiply two bf16 square matrices with dimensions ranging from 1024 to 65536. On Perlmutter, the highest sustained flop/s for matrices of dimensions of 32768×32768 is 280 Tflop/s (90% of peak). On Frontier, the highest sustained flop/s is 125 Tflop/s on 1 GCD (65% of peak) for the same matrix dimensions. For Alps, we referred to a GH200 benchmark guide from NVIDIA that reported a sustained performance of 813 Tflop/s (82% of peak). These numbers show that the vendor advertised peak performance is almost always not achievable in practice. In our evaluation, we also report the % of peak empirical performance achieved by our implementation using the numbers mentioned above.

5.3 Performance Results

We now discuss the results of our performance benchmarking experiments described in Section 5.2.

5.3.1 Weak Scaling Performance

We first present the weak scaling performance of AxoNN on Perlmutter, Frontier and Alps using GPT-style transformers as the application in Figure 5.5 (left). We observe that on all three systems, AxoNN achieves near-ideal weak scaling up to 4096 GPUs/GCDs. This is particularly promising because most large-scale LLM training falls within this hardware range. When running the 60B model on 6144 H100 GPUs of Alps, we see a small reduction in efficiency – 76.5% compared to the performance on 1024 GPUs.

Since Frontier has a significantly large number of GPUs than the other two platforms, we scaled AxoNN on Frontier to 32,768 GCDs. We see near perfect weak scaling up to 8,192 GCDs with a significantly high efficiency of 88.3% (compared to the performance on 512 GCDs). Although our weak performance drops at 16,384 GCDs, we are still able to sustain an efficiency of 79.02%. However, with rising overheads of communication, there is a notable decline in our performance on 32,768 GCDs, and a corresponding drop in efficiency to 53.5%.

We used timers to gather breakdowns of the time per batch into computation and non-overlapped communication to better understand the impact of the performance optimizations described in Section 5.1. We present these results in Figure 5.5 (right), for some model sizes running on 512–8,192 GCDs of Frontier. As a baseline, we use a configuration of AxoNN that corresponds to a hybrid of 1D tensor parallelism within node (similar to Megatron-LM [39]) and

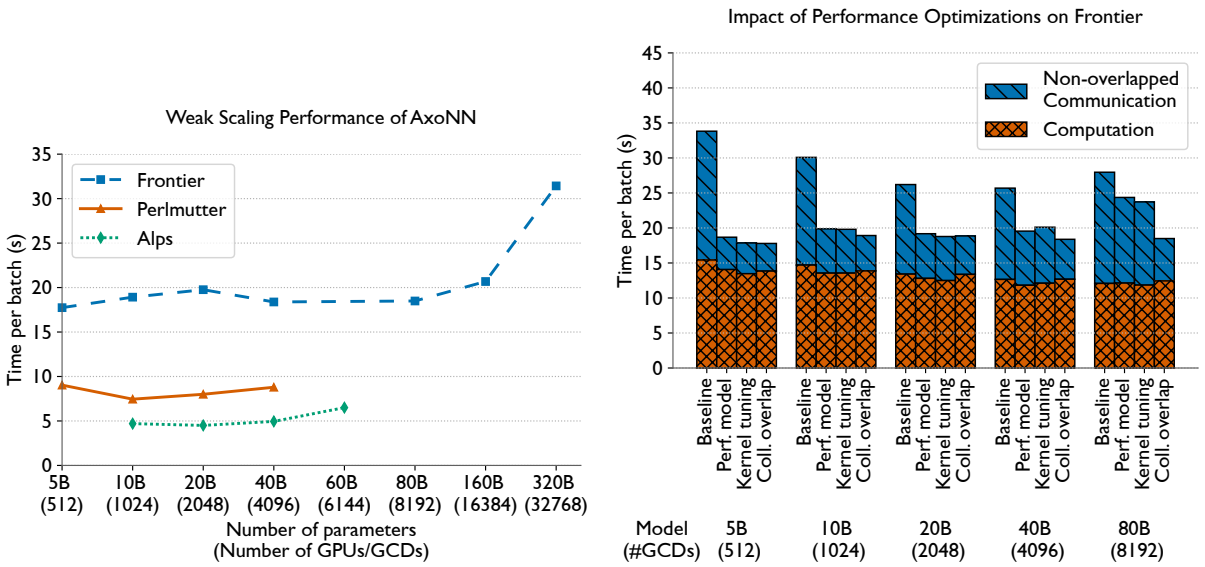

Figure 5.5: Weak scaling performance (time per batch or iteration) of AxoNN on Frontier, Perlmutter, and Alps for models with 5 to 320 billion parameters (left) and the impact of our performance optimizations (right). For the bars labeled "Perf model", we use the best out of the top-10 configurations suggested by our communication model. For the bars labeled "Kernel Tuning" and "Coll Overlap", we enable our matrix multiplication tuning and communication overlap optimizations.

hybrid sharded data parallelism across nodes (similar to FSDP [25, 84]).

We observe that using the 3D parallel matrix multiplication and performance model to select the best configuration results in significant performance improvements of 13-45% over the baseline. Most of the improvement comes from a significant reduction in communication times. For the models in the plot, the improvements in the batch times due to our BLAS kernel tuning are relatively modest (2–4%). Finally, the improvement from our overlap optimizations is largest for the largest model in this series i.e. 80B on 8192 GCDs. In this case, we observe a 22% reduction in the batch times! This is expected because the overheads of communication tend to increase with scale and subsequently the benefits of our overlap optimizations become more pronounced.

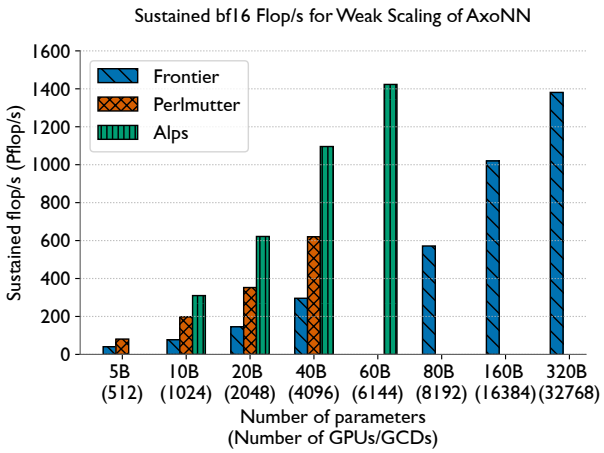

Figure 5.6: Sustained flop/s on different platforms. The FLOP count is calculated analytically for all the matrix multiplication kernels in the code.

5.3.2 Sustained floating point operations per second (flop/s)

Next, we examine the floating-point operations per second (flop/s) achieved by AxoNN. In Figure 5.6, we present the total bf16 flop/s sustained by AxoNN in our weak scaling experiments on Perlmutter, Frontier and Alps. In Table 5.2, we also show our sustained flop/s as a percentage of the vendor advertised and empirical obtained peak flop/s. As discussed in Section 5.2.2, we use 280 Tflop/s, 125 Tflop/s, and 813 Tflop/s as the empirical peak bf16 flop/s for an A100 GPU, an MI250X GCD and an H100 GPU respectively.

On Perlmutter, we observe that AxoNN consistently sustains 50% or higher fraction of the advertised peak of 312 Tflop/s per GPU. As a result of our near perfect weak scaling, we observe that the sustained flop/s also increase linearly from 80.8 Pflop/s on 512 GPUs by nearly $8\times$ to 620.1 Pflop/s on 4096 GPUs. Since the advertised and empirical peak bf16 flop/s of an A100 GPU are close (312 vs. 280 Tflop/s), our % flop/s numbers are also in the same ball park.

On Frontier, in the 512 to 4,096 GCD range, AxoNN achieves near-perfect weak scaling in terms of sustained flop/s which translates to a throughput of around 40% of the advertised

Table 5.2: Sustained flop/s for weak scaling on Perlmutter, Frontier and Alps.

	# GPUs / GCDs	Model	Total Pflop/s	% of Advertised Peak	% of Empirical Peak
Perlmutter	512	5B	80.8	50.6	56.2
	1024	10B	197.8	61.9	68.8
	2048	20B	352.5	55.2	61.3
	4096	40B	620.1	48.5	53.9
Frontier	512	5B	40.4	41.1	63.3
	1024	10B	77.3	39.3	60.4
	2048	20B	145.7	37.0	57.0
	4096	40B	295.9	37.6	57.9
	8192	80B	571.4	36.3	56.0
	16384	160B	1019.9	32.4	49.9
	32768	320B	1381.0	22.0	33.8
Alps	1024	10B	310.0	30.6	37.3
	2048	20B	621.6	30.7	37.4
	4096	40B	1095.8	27.0	33.0
	6144	60B	1423.1	23.4	28.6

peak performance. Notably, this is a significant improvement over Yin et al. [96] and Dash et al. [97] – they achieved a peak of only 30% in a similar range of GCDs, model sizes, and batch sizes on Frontier. AxoNN continues to scale well up to 8,192 GCDs, sustaining 36.3% of the peak and 571.4 Pflop/s in total. Beyond this scale, we start observing scaling inefficiencies. On 16,384 GCDs, we achieve 32.4% of the peak, which amounts to 1.02 Exaflop/s in total. Finally on 32,768 GCDs, our performance drops to 22% of the peak and a total flop/s of 1.381 Exaflop/s. In Section 5.2, we mentioned a significant difference between the advertised peak and the empirically measured peak on a single MI250X GCD (192 vs. 125 Tflop/s). As a result, there is a large difference between AxoNN's flop/s expressed as a percentage of the advertised peak versus the empirical peak. For instance on 32,768 GCDs, these numbers are 22.0% and 33.8% respectively.

On Alps, we observe a similar trend as Perlmutter, with AxoNN consistently sustaining ~30% of the advertised peak up to 4096 GPUs. At 6144 GPUs, we see a slight drop to 23.42% of peak. At 6144 GPUs, we achieve our highest sustained flop/s of 1423.10 Pflop/s across all three machines.

5.3.3 Predicted Time-to-solution

The training of state-of-the-art large language models (LLMs) presents a significant computational challenge due to two key factors. First, the models themselves are large, with current state-of-the-art models comprising hundreds of billions of trainable parameters. Second, LLMs are trained on massive and continually expanding text corpora, often containing trillions of tokens. In this section, we show how AxoNN can significantly reduce the time-to-solution of training such state-of-the-art LLMs on large text corpora. To demonstrate this, we pick the 80B and 640B parameter GPT models from Table 5.1 and collect the per iteration times at various GCD counts. We run the 80B model on 128 to 8,192 GCDs on Frontier, and the 640B model on 512 to 8,192 GCDs. We then extrapolate the batch times to estimate the time it would take to rain these models to completion i.e. to ingest two trillion tokens. These time-to-solution results are presented in Figure 5.7. Note that both the model size and the number of tokens are representative of modern LLM training setups such as Meta's Llama [13].

As the plot shows, training an 80B model on 128 GCDs will take 50 months or more than four years. This emphasizes the critical role of large-scale parallelism in LLM training. As we scale to more GCDs, we see the expected time to solution drop almost linearly till 8,192 GCDs. Our estimate for the total training time of the 80B model on 8,192 GCDs is a much more

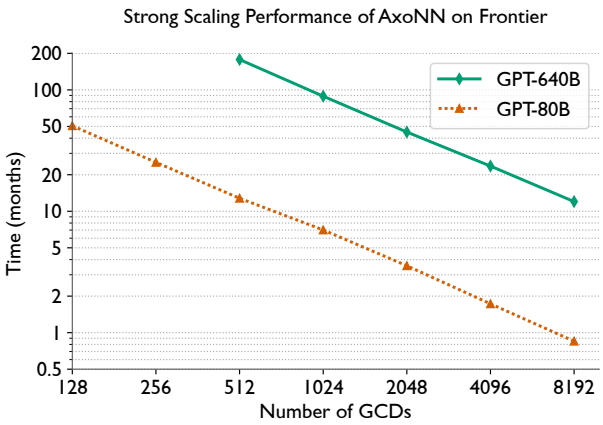

Figure 5.7: Strong scaling showing expected time-to-solution on Frontier. Using the average time per iteration, we predict the training times for GPT-80B and GPT-640B on 2T tokens for various GCD counts.

reasonable 25.5 days. For the 640B model, even a much larger GCD count of 512 GCDs is impractical, with the estimated time-to-solution amounting to 14 years. However, on 8,192 GCDs, the estimated total training time is 15 months, which is an 11$\times$ improvement. For both models, this amounts to a strong scaling efficiency of more than 90%. These experiments underscore AxoNN's efficacy in significantly reducing the pre-training time for cutting-edge LLMs trained using massive datasets. By enabling faster training cycles on large scale multi-GPU clusters such as Frontier and Alps, AxoNN has the potential to accelerate the overall pace of LLM research and development.

Chapter 6: Optimizing Collectives with Large Payloads on GPU-based Supercomputers

In the preceding chapters of this dissertation, we have assumed that the underlying communication libraries are highly optimized, allowing us to focus on minimizing communication overhead at the framework level. This assumption was made explicit in Chapter 5, where our communication model idealized a scenario with zero latency and perfect bandwidth utilization (see Section 5.1.2). However, real-world performance is far from ideal, and understanding the true efficiency of communication libraries is crucial for scaling distributed deep learning workloads. In this chapter, we go beyond performance evaluation to systematically analyze Cray-MPICH and RCCL, the primary communication backends available on the Frontier supercomputer. Our focus is on two key collectives – all-gather and reduce-scatter – which are critical to state-of-the-art distributed deep learning frameworks such as FSDP [25], ZeRO [23], and AxoNN's z-dimension in 3D parallel matrix multiplication (see Chapter 5). Through this analysis, we identify several bottlenecks that hinder these libraries from efficiently scaling to thousands of GPUs. Finally, we develop highly optimized implementations of these collectives that significantly improve performance, thereby enabling more efficient large-scale distributed training on modern supercomputing architectures.

6.1 Identifying Issues with Cray MPICH and NCCL/RCCL

As established in the previous section, all-gather and reduce-scatter operations are important collectives in parallel deep learning [23, 25, 83]. Thus, to scale model training to the thousands of GPUs required for large models, we need highly efficient and scalable implementations of these collectives – particularly for the large message sizes characteristic of deep learning workloads (see Figure 6.1). This section investigates the state of the current state of the practice of popular communication libraries - Cray-MPICH and RCCL, for these collectives. We find unique issues that plague the performance of each library, and we highlight these below via experiments on the Frontier supercomputer.

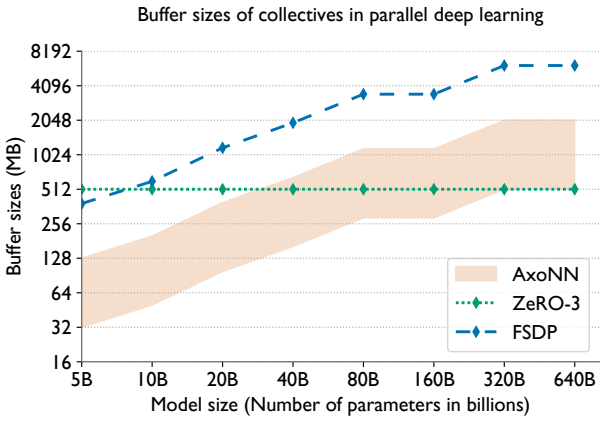

Figure 6.1: Distribution of all-gather and reduce-scatter message sizes for several deep learning frameworks for a range of transformer [8] model sizes. The y-axis represents input buffer sizes for all-gathers but output buffer sizes for reduce-scatters.

6.1.1 Benchmarking Methodology

First, let us look at the methodology we used to benchmark the performance of the two libraries on Frontier.

Process Placement and NUMA Configuration: Each node on Frontier is equipped with four GPUs, which are each partitioned into two Graphic Compute Dies (GCDs). We therefore launch one MPI process per GCD, binding each process to seven CPU cores and leaving two cores per NUMA region free for operating-system tasks and to minimize noise. To ensure NUMA-aware placement of network interfaces, we configure the OFI provider with `MPICH_OFI_NIC_POLICY=USER` and `MPICH_OFI_NIC_MAPPING="0:0-1;1:2-3;2:4-5;3:6-7"`, which pins each MPI rank to the appropriate NIC ports in its NUMA domain.

Message Sizes and Measurement Protocol: In line with Figure 6.1, our evaluation focuses on message sizes from 16MB up to 1GB. Note that for all-gathers and reduce-scatters, these values refer to the output and input message size per GPU respectively. For each combination of library, collective, message size, and GPU count, we perform ten independent runs. We measure the total time spent in the collective on each run using AMD's `hipeventtimers` instrumentation and computed the mean and standard deviation over the ten runs to ensure statistical robustness.

Communication Tuning: We disable all forms of eager messaging in the OFI/CXI provider by setting `FI_CXI_RDZV_THRESHOLD=0`, `FI_CXI_RDZV_GET_MIN=0`, and `FI_CXI_RDZV_EAGER_SIZE=0`. For the message sizes we target, disabling eager messaging significantly improves collective performance for both Cray MPICH and RCCL. We enable GPU Direct RDMA for GPUs and NICs sharing the same NUMA node via `NCCL_NET_GDR_LEVEL=PHB` and disable HSA's SDMA engine by setting `HSA_ENABLE_SDMA=0`, ensuring that data transfers bypass the host.

Software Stack: Our software stack comprises of ROCm 6.2.4, RCCL 2.20.5, Cray MPICH 8.1.31 distribution, libfabric 1.15.2 and the aws-ofi-rccl plugin version v1.4.

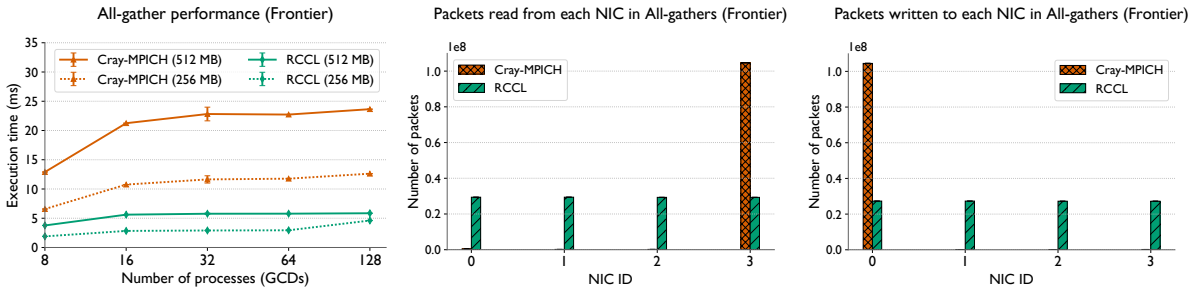

Figure 6.2: The left plot compares all-gather performance of Cray MPICH and RCCL on Frontier for a bandwidth-bound scenario with large message sizes (256 and 512 MB) and small GPU counts. The middle and right plot show the number of packets read from (left) and written to (right) each of the four NICs on a Frontier compute node during all-gather operations.

6.1.2 Poor MPI performance at lower GPU counts

Figure 6.2 (left) presents a comparative analysis of all-gather performance between Cray MPICH and RCCL on Frontier, specifically for large message sizes of 256 MB and 512 MB. Despite both libraries implementing a ring-based collective algorithm over the OpenFabrics Interfaces (OFI) layer, RCCL achieves approximately a $4\times$ performance advantage in this bandwidth-bound scenario. To explain this disparity, we examine hardware performance counters provided by the Cassini Slingshot-11 Network Interface Controllers (NICs) [98] on each Frontier node.

Our investigation focuses on the counters `parbs_tarb_pi_posted_pkts` and `parbs_tarb_pi_non_posted_pkts`, which, based on our understanding, represent the count of packets read from and written to each NIC within a node during job execution. The middle and right plots of Figure 6.2 demonstrate a significant divergence in NIC utilization between the two libraries. Cray MPICH constrains all read operations to NIC 3 and all write operations to NIC 0, effectively creating a single-NIC bottleneck. Conversely, RCCL distributes network traffic more uniformly across all available NICs. This equitable load distribution leads to enhanced bandwidth utilization and a substantial reduction in all-gather execution time. This observed imbalance in

NIC utilization directly accounts for the pronounced performance gap between Cray MPICH and RCCL in this bandwidth-bound context.

Explanation 1

Cray MPICH routes all network traffic through a single NIC, resulting in severe underutilization of the available network bandwidth. In contrast, RCCL effectively balances node traffic across all four NICs, achieving a four-fold performance improvement over Cray MPICH.

6.1.3 Poor Performance of MPI_Reduce_scatter

Next, we examine reduce-scatter performance. As shown in Figure 6.3, Cray MPICH (orange) performs significantly worse than RCCL (green). Notably, this performance gap is far greater than the $4\times$ difference observed earlier for all-gather in Figure 6.2. While the NIC underutilization issue outlined in the previous subsection still persists for Cray-MPICH Reduce-scatters, it alone cannot explain this performance disparity. We hypothesize that this disparity stems from the way reduction computations are scheduled in the two libraries. Cray MPICH performs the reduction operations required for reduce-scatter on the CPU, introducing significant computational overheads for large messages. In contrast, RCCL efficiently performs these operations by offloading them to the GPUs, leveraging their parallel processing capabilities.

To test this hypothesis, we manually implemented the reduce-scatter operation using Cray MPICH point-to-point sends and receives, while scheduling the reduction operations on the GPU via a HIP vector-addition kernel. As shown in Figure 6.3, our implementation (blue line) achieves performance that is several times faster than Cray MPICH's native reduce-scatter, further supporting our hypothesis.

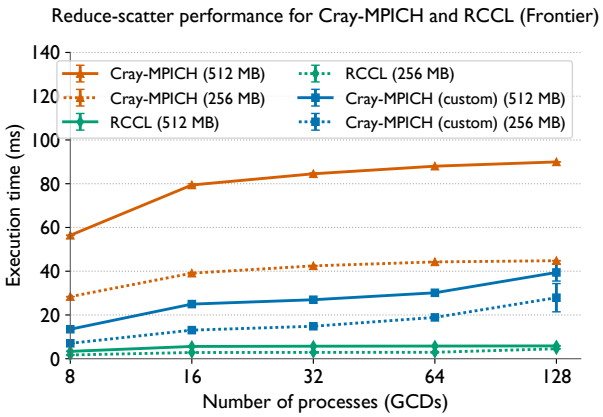

Figure 6.3: Performance comparison of reduce-scatter using Cray MPICH, RCCL, and a custom implementation of reduce-scatter that uses Cray MPICH P2P and GPU compute kernels.

Explanation 2

Cray MPICH's CPU-based reduction operations in reduce-scatter introduce significant overhead, which in combination with the NIC underutilization issue, results in a 10-15x performance gap compared to RCCL's GPU-accelerated reductions.

6.1.4 Poor Scaling of RCCL and MPI at Large GPU Counts

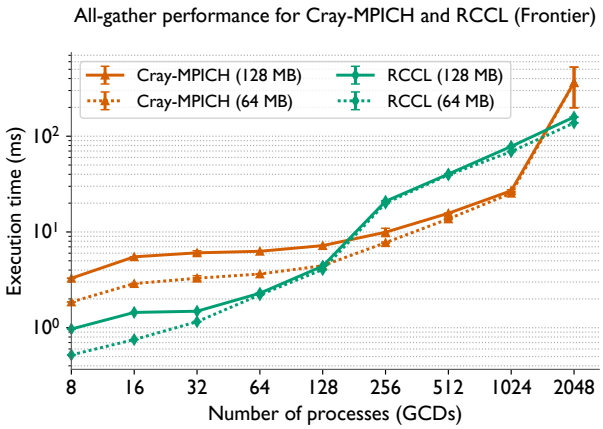

Figure 6.4: Performance comparison of all-gather using Cray-MPICH vs. RCCL on Frontier for two output buffer sizes of 64 and 128 MB. The ideal scaling behavior (flat horizontal line) is not achieved by either library, highlighting their limited scalability at increasing GCD counts.

Figure 6.4 shows all-gather performance for Cray MPICH and RCCL when sending relatively small messages across increasing numbers of GCDs. We observe that both libraries exhibit poor scaling behavior at large GPU counts. On investigating deeper, we found that both Cray MPICH and RCCL only support the ring algorithm for all-gathers and reduce-scatters. While effective for bandwidth-bound workloads, the ring algorithm performs poorly in latency-bound scenarios because each process must send and receive $(p-1)$messages sequentially, causing the total communication time to grow linearly with the number of processes.

Curiously, neither library implements more optimal algorithms like recursive doubling or halving, which are known to reduce the number of communication steps to $\log_2 p$ and are generally preferred for small message sizes or high process counts. This lack of algorithmic diversity directly contributes to the sub-optimal scaling we observe at large GPU counts.

Explanation 3

Both Cray MPICH and RCCL rely solely on the ring algorithm for all-gather and reduce-scatter, leading to poor scaling inlatency-bound scenarios. More efficient algorithms like recursive doubling and halving are not supported.

6.2 Optimizing All-gathers and Reduce-scatters

In Section 6.1, we identified several challenges affecting RCCL and Cray-MPICH in the context of all-gather and reduce-scatter collectives for deep learning workloads. These challenges create significant barriers to efficiently scaling large model training across thousands of GPUs. The central theme of this work is to developed optimized implementations of all-gather and reduce-scatter collectives that address these challenges. In this section, we present our pro-

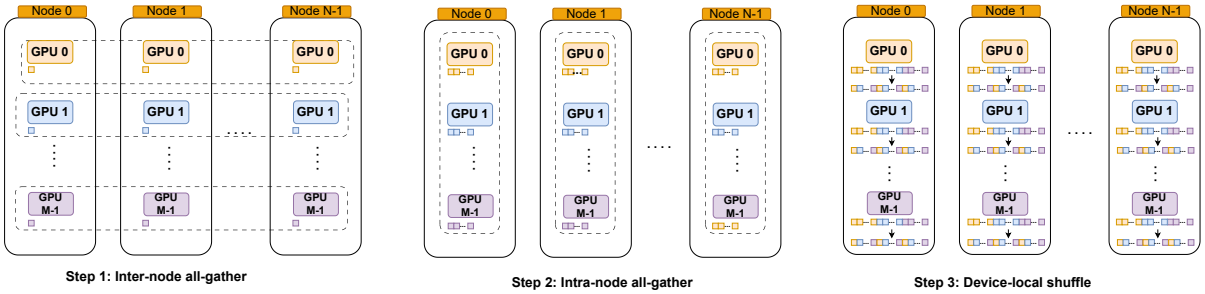

Figure 6.5: Diagram showing our hierarchical (two-level) implementation to dissolve an all-gather operation on a GPU-based cluster with N nodes and M GPUs per node. In Step 1, we performs inter-node all-gathers, in step 2, we perform intra-node all-gathers and in step 3, each GPU performs a local shuffle of the received data.

posed solutions, which are implemented in a new library called PCCL (Performant Collective Communication Library). We begin by discussing the design principles and strategies that drive our proposed solutions.

6.2.1 Hierarchical Collective Algorithms for Load Balancing NIC Traffic

Our optimized implementations of all-gather and reduce-scatter are based on a two-level hierarchical design. While prior work has demonstrated that hierarchical algorithms can reduce latency and improve scalability in collective operations [66,68], our primary motivation for adopting this design is to address the NIC underutilization problem identified in Section 6.1.2. Now, we provide a brief overview of the inner workings of our hierarchical design.

We illustrate our design in Figure 6.5 for an all-gather operation on a hypothetical system with N nodes and M GPUs per node. The global collective operation is divided into two distinct phases using sub-communicators: inter-node sub-communicators and intra-node sub-communicators. Inter-node sub-communicators are formed by grouping corresponding GPUs across nodes in a group. For example, in Figure 6.5, all GPUs with the same index across nodes are grouped together to form a total of M inter-node sub-communicators. Similarly, intra-node

sub-communicators are formed by grouping together all GPUs within a node.

The hierarchical communication unfolds in three steps. First, in the inter-node all-gather phase, we schedule an all-gather operation in all of the inter-node sub-communicators. This is illustrated in Step-1 of Figure 6.5. Once this phase is completed, each GPU in a node has received data from its corresponding GPU in the other nodes. Now, within every node we have the entire result of the all-gather operation, but the data is split across GPUs. Therefore, the next step is to perform an intra-node all-gather operation, which is illustrated in Step-2 of Figure 6.5. Once this phase is complete, each GPU now has the complete output in its memory, albeit in an incorrect order. So, the final step involves a device-local shuffle operation, where each GPU rearranges its data to put in a correct order. This is illustrated in Step-3 of Figure 6.5. The device-local shuffle is performed using a transpose kernel in practice. We implement reduce-scatter in a similar manner – but starting with the intra-node phase first, followed by the inter-node phase.

Having explained the workings of our hierarchical design, let us now see how it addresses the NIC underutilization problem. An important aspect of our design in that it schedules all of the all-gather operations in Step-1 of Figure 6.5 concurrently on all of the inter-node sub-communicators. We leverage this fact to utilize all NICs on a node concurrently. A Frontier node has four NICs, each connected to two GCDs. In our implementation, we ensure that each GCD exclusively sends and receives traffic to and from its corresponding NIC (e.g. - GCDs 0 and 1 to NIC 0, GCDs 2 and 3 to NIC 1, and so on). This is how we ensure that the inter-node traffic is evenly distributed across all NICs in PCCL.

6.2.2 Choice of Communication Libraries for Each Level of the Hierarchy

We now describe the choice of communication libraries we make for each level of the hierarchy, starting with the intra-node level. GPU-vendor libraries like RCCL are highly optimized for intra-node topologies, efficiently utilizing shared memory, PCIe, and Infinity fabric connections. These optimizations significantly outperform most MPI implementations in managing GPU-to-GPU communication within a node [65]. Hence, we simply rely on RCCL for all of our communication in the intra-node phase.

Prior work has reported that RCCL is not robust at scale and can crash during training runs [99]. This issue has also been noted by HPE[1] and in the OLCF User Guide[2]. Thus for reliability reasons, we opt to use Cray-MPICH for all inter-node communication.

6.2.3 Choice of Algorithms for Inter-Node Communication

Our choice of communication algorithms for each level of the hierarchy is driven by performance considerations and the limitations of available libraries. Since RCCL only supports the ring algorithm for intra-node collectives, we adopt this as our intra-node communication strategy. Fortunately, ring is well-suited for this context, as the small number of GCDs within a node (eight) ensures that ring can effectively saturate the available bandwidth.

The inter-node phase, however, presents greater challenges. With potentially thousands of GPUs participating in the collective, latency concerns become critical. Cray-MPICH, which we rely on for inter-node communication due to RCCL's instability at scale, offers only the ring

[1] https://www.olcf.ornl.gov/wp-content/uploads/OLCF_AI_Training_0417_2024.pdf

[2] https://docs.olcf.ornl.gov/software/analytics/pytorch_frontier.html#environment-variables

algorithm by default. Unfortunately, ring algorithms's linear scaling in latency with respect to the number of processes makes it suboptimal at large-scale.

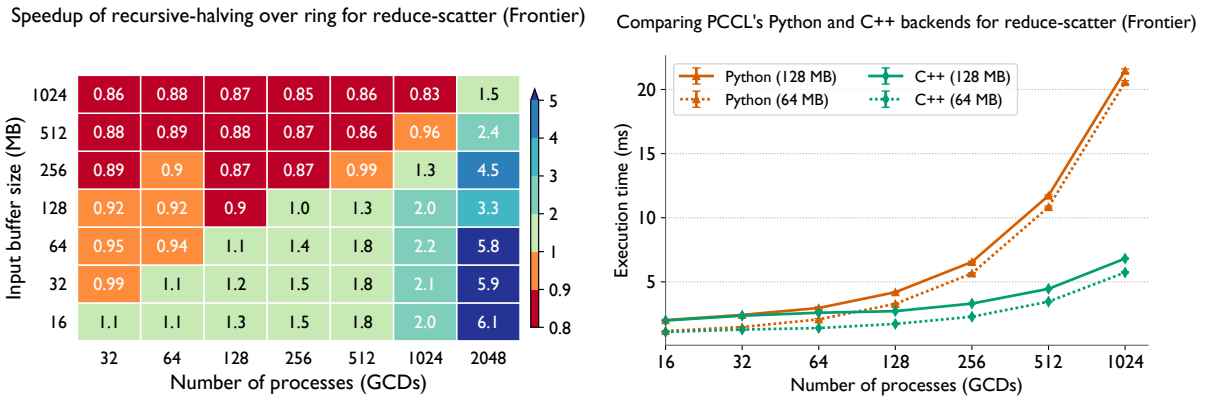

Figure 6.6: (Left) Heatmap showing speedups from using recursive halving over the ring algorithm in the inter-node phase of the reduce-scatter implementation in PCCL, and (Right) Performance comparison of the C++ (with Pybind11) and Python based implementations of reduce-scatter in PCCL.

To address this, we implement alternative algorithms with improved scaling properties. Specifically, we utilize recursive doubling for all-gather operations and recursive halving for reduce-scatter operations [60]. These algorithms offer logarithmic latency terms, enabling significantly better performance as the number of GPUs increases. Our implementations are based on Cray-MPICH's point-to-point send and receive operations. Moreover, for reduce-scatter operations, we also ensure that our vector addition computation is efficient by scheduling it on the GPU cores.

In Figure 6.6 (left), we demonstrate the speedup of using recursive halving over ring in the inter-node phase of our reduce-scatters. Note that both of these implementations use RCCL's ring algorithm for the intra-node phase. We observe that ring is the preferred choice of algorithm for inter-node communication in bandwidth bound scenarios (smaller process counts and/or larger message sizes). However, as expected recursive halving becomes the more optimal algorithm for

latency bound scenarios (larger process counts and/or small message sizes).

We develop these implementations in C++ as part of PCCL and expose Pybind11 bindings to enable seamless integration with Python-based deep learning frameworks such as ZeRO-3 [23]. Implementing these algorithms in C++ proves to be critical for achieving high performance. As shown in Figure 6.6 (right), a baseline implementation using Python and mpi4py can be nearly $4\times$ slower than our optimized C++ version—compare the performance at 1024 GCDs. This highlights the importance of minimizing CPU-side overhead and reducing language-level inefficiencies for large-scale collective communication.

6.3 Performance Results

We now present and analyze the results of the empirical experiments we conducted to evaluate the efficacy of PCCL for scaling parallel training.

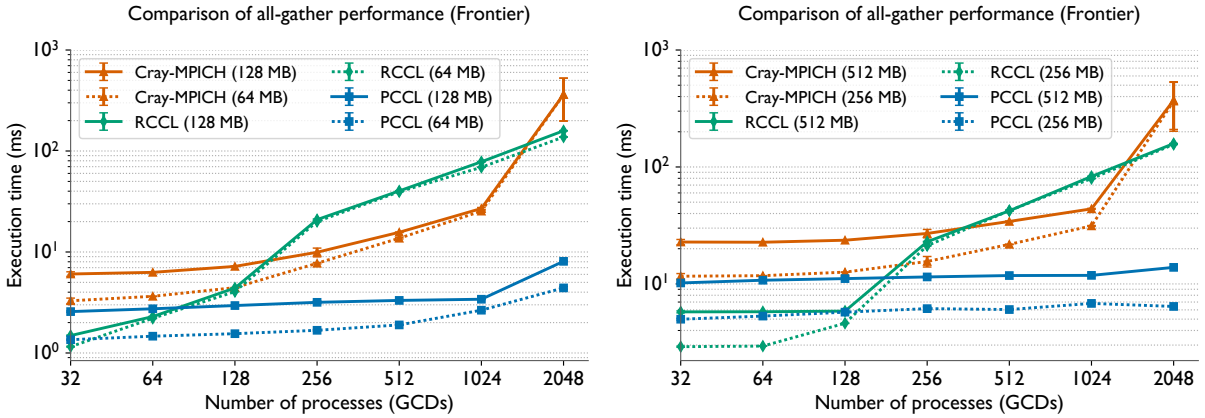

Figure 6.7: Performance comparison of all-gather using Cray MPICH, RCCL, and PCCL, for different per-process output buffer sizes (left plot: 64 and 128 MB, right plot: 256 and 512 MB) and varying process counts on Frontier.

6.3.1 Performance Improvements Using PCCL

We begin by examining the performance of PCCL for all-gather and reduce-scatter operations and comparing it against other state-of-the-art communication libraries.

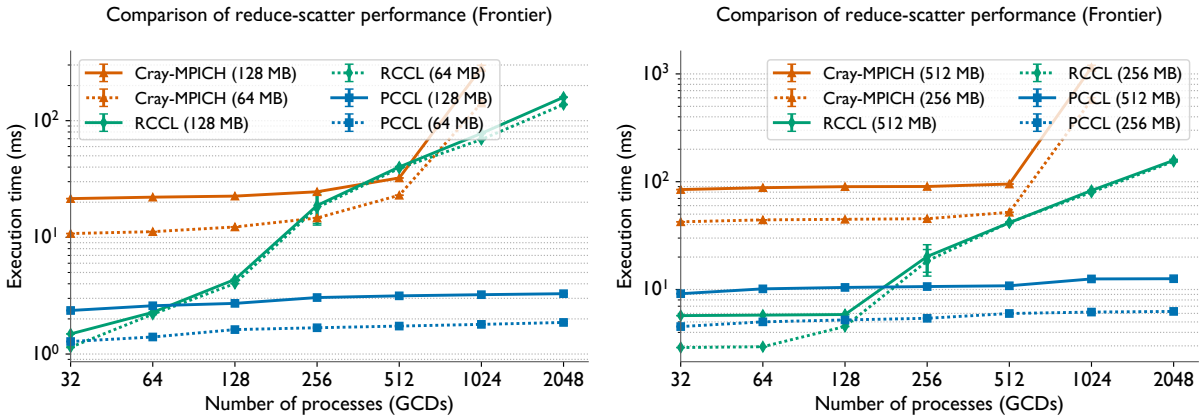

Figure 6.8: Performance comparison of reduce-scatter using Cray MPICH, RCCL, and PCCL, for different per-process input buffer sizes (left plot: 64 and 128 MB, right plot: 256 and 512 MB) and varying process counts on Frontier.

6.3.1.1 Comparison with Cray-MPICH and RCCL on Frontier

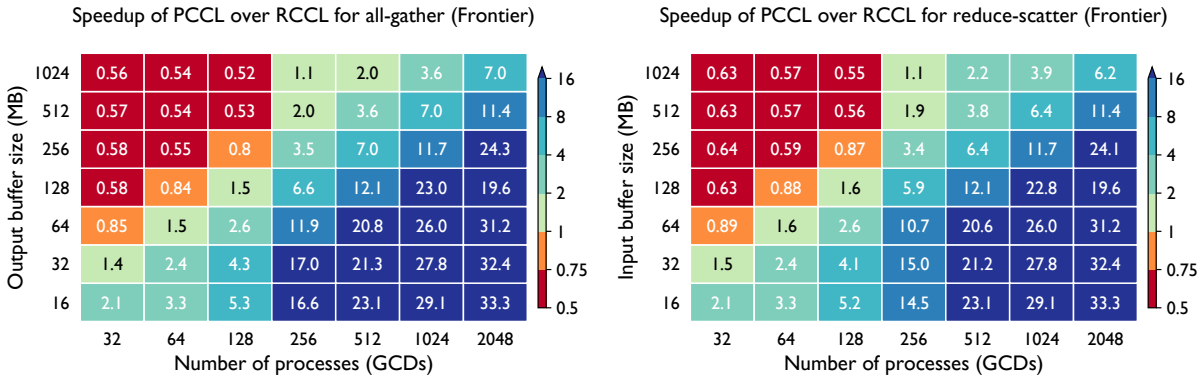

Figure 6.9: Heatmaps showing speedups from using PCCL over RCCL for all-gather (left) and reduce-scatter (right) on Frontier. The speedup is shown as a function of per-process output/input buffer size (in MB) and process count.

Let us start with examining PCCL's performance on the Frontier supercomputer. Figure 6.7

shows the performance of all-gather operations on Frontier using PCCL and other communication libraries. We evaluate two sets of output buffer sizes: 64 and 128 MB (left plot), and 256 and 512 MB (right plot). For each configuration, we scale the number of GCDs from 32 to 2048. Since the output buffer size per GPU remains fixed, the ideal performance curve for each buffer size is a flat horizontal line, indicating perfect scaling.

However, we observe that RCCL and Cray-MPICH fall short of this ideal. For smaller message sizes in the left plot, RCCL (green lines) scales poorly, with execution time increasing almost linearly with the number of processes. For the larger message sizes in the right plot, RCCL performs well up to 128 processes, but experiences significant degradation beyond that—mirroring the trends seen with smaller messages. Cray-MPICH (orange lines) shows a similar pattern, with performance dropping sharply as we scale to higher process counts. We attribute the poor scaling of RCCL and Cray-MPICH to their reliance on the ring algorithm, whose latency term grows linearly with the number of processes.

In contrast, PCCL (blue lines) maintains nearly flat scaling trends across all message sizes in both plots, demonstrating significantly better scalability and efficiency. We attribute PCCL's better performance to its hybrid strategy, described in Section 6.2, which exploits the heterogeneous network topology. By using the ring algorithm within nodes (limited to eight processes) and recursive doubling across nodes, PCCL bounds the latency overhead that otherwise grows linearly in traditional ring-based implementations. This design enables better scalability across large GPU counts. The performance improvements of PCCL over RCCL and Cray-MPICH become increasingly pronounced as we increase the number of processes (GCDs). At 2048 processes, PCCL achieves speedups ranging from 7 – $24\times$ over RCCL, and an even larger 27 to $82\times$ over Cray-MPICH, depending on the message size. These results highlight PCCL's ability

to deliver high-performance communication at scale.

Next, let us examine reduce-scatter performance on Frontier, as shown in Figure 6.8. Again, we observe similar trends as in the case of all-gather. Both RCCL and Cray-MPICH fall short of the ideal performance curve, and PCCL acheives significant speedups over both libraries with increasing scale.

Since RCCL is the default library used by most distributed deep learning applications on AMD platforms, let us take a closer look at how PCCL compares against it. Figure 6.9 shows the speedups of PCCL over RCCL for all-gather (left) and reduce-scatter (right) operations on Frontier, respectively, across a range of output buffer sizes and process counts. In the top-left regions of both heatmaps–large messages and small GPU counts, which represent bandwidth-bound scenarios–PCCL underperforms RCCL. For instance, with a 1024 MB buffer at 32 GCDs, speedups are around $0.52\times$ for all-gather and $0.55\times$ for reduce-scatter. This is expected, as RCCL's flat ring algorithm can theoretically achieve higher bandwidth than PCCL's hierarchical two-phase strategy [66].

However, in the bottom-right corners–small messages and large GPU counts, where latency dominates–PCCL delivers substantial gains. For both collective operations at 2048 GCDs, PCCL achieves speedups of more than $30\times$ over RCCL for 16MB, 32MB, and 64MB message sizes, respectively! In contrast, for larger message sizes like 512MB and 1024MB, the speedups are smaller but still significantly high —- 11.4 and $7\times$ for all-gather, and 11.4 and $6.2\times$ for reduce-scatter. These results underscore PCCL's strength in latency-bound scenarios and highlight its ability to scale efficiently to thousands of GPUs.

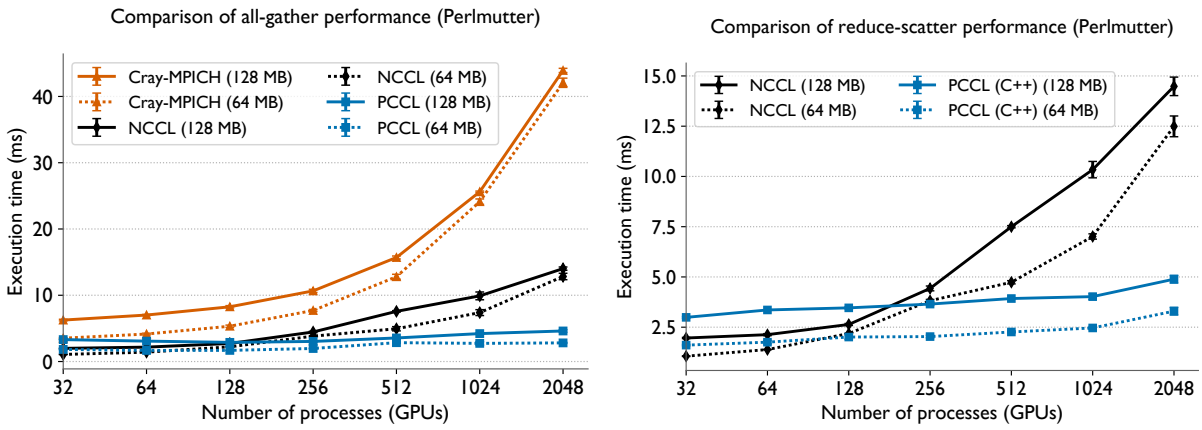

Figure 6.10: Performance comparison of all-gather (left plot) and reduce-scatter (right plot) using Cray MPICH, NCCL, and PCCL, for two per-process buffer sizes (64 and 128 MB) and varying process counts on Perlmutter.

6.3.1.2 Comparison with Cray-MPICH and NCCL on Perlmutter

We now evaluate PCCL's effectiveness on Perlmutter, which features NVIDIA A100 GPUs. Figure 6.10 presents results for all-gather (left) and reduce-scatter (right) operations with message sizes of 64MB and 128MB–representing the output buffer sizes for all-gather and input buffer sizes for reduce-scatter, respectively. We observe similar trends as on Frontier. Both Cray-MPICH (orange lines) and NCCL (black lines) fall short of ideal scaling, which would appear as a flat horizontal line. Like RCCL on Frontier, NCCL's performance begins to degrade noticeably beyond 128 processes. In contrast, PCCL scales nearly perfectly across both collectives, maintaining desirable flat performance curves and achieving speedups in the range of $1.3 - 4.6\times$ over NCCL and $8.8–15\times$ over Cray-MPICH on 1024 and 2048 GPUs!

Figure 6.11 examines how PCCL compares to NCCL across various message sizes and GPU counts. Similar to RCCL, NCCL outperforms our library in the top-left regions of the heatmaps, representing bandwidth-bound scenarios. For instance, with 32 processes and a 1024 MB message size, NCCL is nearly $1.5\times$ faster than PCCL. However, as we transition to latency-

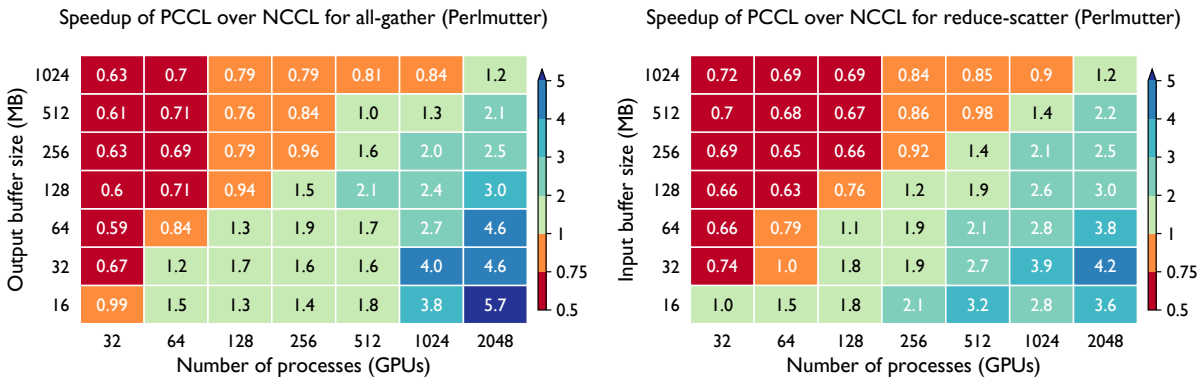

Figure 6.11: Heatmaps showing speedups from using PCCL over NCCL for all-gather (left) and reduce-scatter (right) on Perlmutter. The speedup is shown as a function of per-process output/input buffer size (in MB) and process count.

bound regions in the bottom-right corners of the heatmap, PCCL's advantages become evident. Around 1024–2048 processes and 16–32MB message sizes, PCCL achieves significant speedups over NCCL, ranging from 3–5×. While speedups for larger message sizes are smaller, they remain notable. For example, at 2048 processes and 128–512 MB message sizes, PCCL is approximately 2–3× faster than NCCL. These results highlight PCCL's effectiveness in accelerating collective communication for parallel deep learning – across both extreme scales and diverse GPU architectures.

6.3.2 Impact on DL Applications' Performance

Finally, we examine how these communication gains translate into improvements in end-to-end training performance at scale. The left panel of Figure 6.12 presents the batch times for strong scaling GPT-3-style transformer training on Frontier using the DeepSpeed ZeRO-3 framework [23]. Green lines represent ZeRO-3 runs with RCCL, the default communication library and blue lines represent runs with all-gather and reduce-scatter collectives in ZeRO-3 issued with PCCL. At smaller scales (128 and 256 GCDs), both libraries perform comparably.

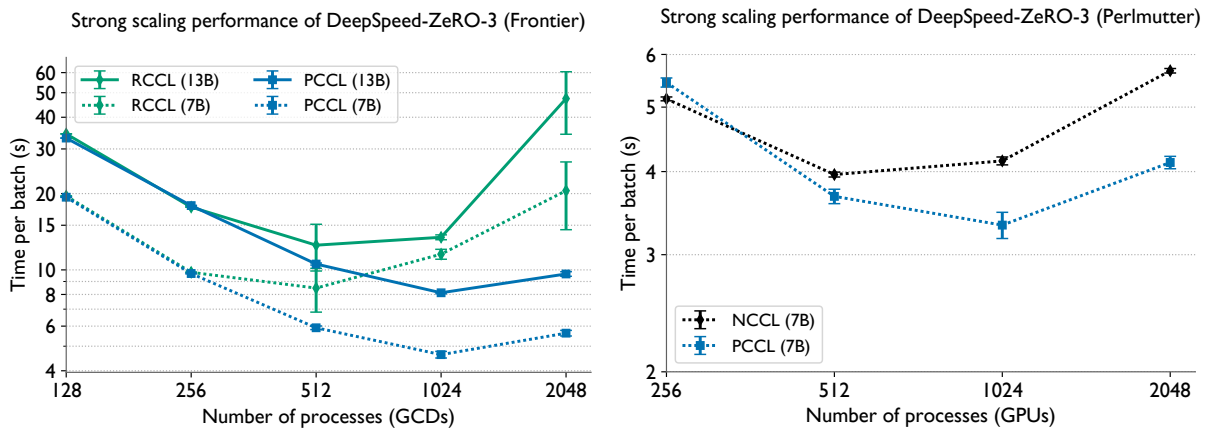

Figure 6.12: Strong scaling performance of Deepspeed ZeRO-3 using RCCL, NCCL, and PCCL, on Frontier (left) and Perlmutter (right) for two model sizes: GPT-3 7B and GPT-3 13B.

However, as we scale further, PCCL begins to outperform RCCL. At 512 GCDs, PCCL reduces batch time by nearly 30% for the 7B model and by 16% for the 13B model. When scaling to 1024 GCDs, RCCL fails to maintain strong scaling and even exhibits increased batch times compared to 512 GCDs. In contrast, PCCL continues to scale efficiently, delivering a 60% speedup for the 7B model and a 39% speedup for the 13B model. Finally, at 2048 GCDs, although both libraries show diminishing returns in strong scaling efficiency, PCCL still achieves substantial speedups (70–80%) relative to RCCL.

We observe similar trends on Perlmutter, as shown in the right panel of Figure 6.12. At 256 GPUs, NCCL outperforms PCCL by approximately 6%. However, as we scale to larger GPU counts, PCCL begins to outperform NCCL–achieving a 7% speedup at 512 GPUs and a significantly higher 20% speedup at 1024 GPUs. All of these results highlight PCCL's ability to deliver performance improvements for collective communication across multiple GPU architectures, and more importantly, translate those gains into meaningful end-to-end speedups for large-scale deep learning applications.

Chapter 7: A Hybrid Tensor-Expert-Data Parallelism Approach to Optimize Mixture-of-Experts Training

Mixture-of-Experts (MoE) is a neural network architecture that adds sparsely activated expert blocks to a base model, increasing the number of parameters without impacting computational costs. However, current distributed deep learning frameworks are limited in their ability to train high-quality MoE models with large base models. This chapter dicusses DeepSpeed-TED [100], a framework featuring a three-dimensional, hybrid parallel algorithm that combines data, tensor, and expert parallelism to enable the training of MoE models with 4–8$\times$ larger base models than the current state-of-the-art.

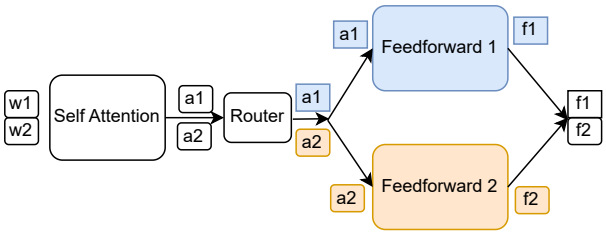

Figure 7.1: A single Mixture-of-Experts (MoE) layer with two "experts" or feedforward blocks. The input batch has two tokens, w1 and w2. We use the prefixes 'w', 'a', and 'f' to denote the input activations to the layer, output activations of self-attention and feedforward blocks respectively. Similarly we label each activation with an integer suffix corresponding to its token. Note that each token is uniquely routed to a single expert by a parameterized routing function.

7.1 TED: A Hybrid Tensor-Expert-Data Parallel Approach

By adding sparsely activated experts, the Mixture-of-Experts architecture allows us to make a given neural network, i.e. the base model, arbitrarily large while keeping its computation cost unchanged. However, merely increasing the number of experts yields diminishing returns in model generalization beyond 64–128 experts [101]. To build high quality MoEs, it is imperative that we increase the base model sizes as well as the number of experts [102]. In this section, we provide an overview of TED, our hybrid parallel approach which combines DeepSpeed-MoE's expert [10], MegatronLM's tensor [39] and ZeRO's data [23] parallelism, to enable the training of such MoEs with extremely large multi-billion parameter base models on multi-GPU clusters. In this work, we use the first stage of ZeRO, which shards the optimizer states across data parallel GPUs. While further stages of their optimizations (stage-2, 3, offload [103] and infinity [28]) can support training of larger models, this happens at a cost to performance.

We use the terms non-expert and expert blocks interchangeably with self-attention and feedforward blocks respectively. Note that TED parallelizes the computation of expert and non expert blocks in a different manner. This is because expert parallelism is only applicable to the feedforward blocks of the transformer base model. Thus, TED uses a two dimensional hybrid of tensor and data parallelism to parallelize the non-expert blocks. Whereas, it utilizes all three of tensor, expert, and data parallelism for the expert blocks. Under TED, we organize available GPUs into two different virtual topologies for the non-expert and expert blocks. We illustrate these topologies in Figure 7.2.

For the non-expert blocks, we maintain a two dimensional (2D) topology of GPUs, one dimension each for tensor and data parallelism. In this topology, GPUs in a row implement

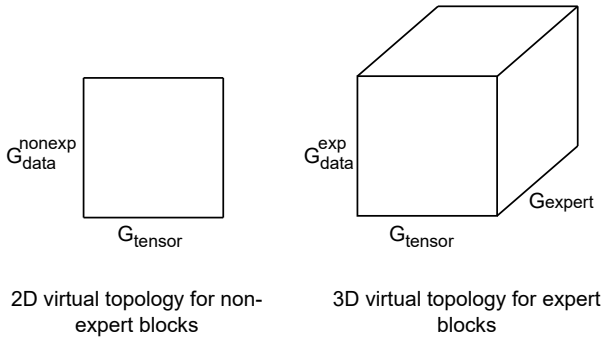

Figure 7.2: TED uses a two dimensional hybrid of tensor and data parallelism to parallelize the computation of non-expert blocks. Whereas, it utilizes all three of tensor, expert, and data parallelism to parallelize expert blocks.

tensor parallelism, and we refer to a row of GPUs as a tensor parallel group. Similarly, TED realizes data parallelism across columns of GPUs, and we refer to these columns as data parallel groups. Likewise, for the expert blocks, we maintain a three dimensional (3D) topology of GPUs, one each for tensor, expert, and data parallelism. To form the tensor parallel groups for the expert blocks, we reuse the tensor parallel groups formed in the 2D topology for the non-expert blocks. However, we further decompose the data parallel groups of the non-expert blocks into a 2D topology to form groups for expert parallelism and data parallelism for the expert blocks. We define G_{tensor} and G_{data}^{nonexp} as the size of the tensor parallel and non-expert data parallel groups respectively. Similarly, we define G_{expert} and G_{data}^{exp} as the size of the expert parallel and expert data parallel groups respectively. Following prior work [23], we always set G_{expert} to the number of experts in the model for performance considerations. Note that given a number GPUs, G, the following relation always holds true:

$$G_{tensor} \times G_{expert} \times G_{data}^{exp} = G_{tensor} \times G_{data}^{nonexp} = G \tag{7.1}$$

In Figure 7.3, we illustrate the forward pass of an MoE layer with two experts on four

GPUs. As mentioned previously, we set G_{expert} to the number of experts i.e. 2. The other degrees of parallelism are $G_{tensor} = 2$, $G_{data}^{nonexp} = 2$, $G_{expert} = 2$, and $G_{data}^{exp} = 1$. We partition the parameters of the self-attention block (non-expert) and the two feed forward blocks (experts) as per the semantics of MegatronLM's tensor parallelism and place the first partition on GPUs 0 and 2 and the second partition on GPUs 1 and 3. GPUs (0,1) and (2,3) thus form the two tensor parallel groups. GPU pairs (0,2) and (1,3) comprise the data parallel groups for the non-expert parameters. The same GPU pairs however comprise the expert parallel groups for the expert parameters. The four GPUs individually form singleton data parallel groups for the expert parameters.

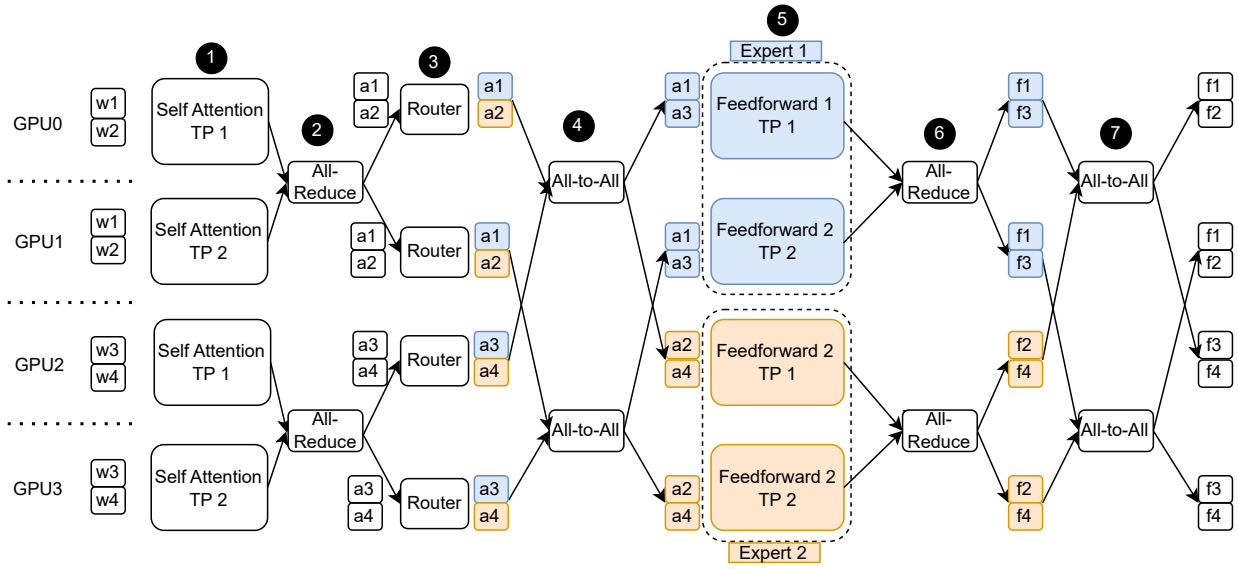

Figure 7.3: Forward pass of an MoE layer with two experts on four GPUs using TED. We use a $G_{tensor} \times G_{data}^{nonexp} = 2 \times 2$ topology for the non-expert self-attention blocks and $G_{tensor} \times G_{expert} \times G_{data}^{exp} = 2 \times 2 \times 1$ topology for the expert feedforward blocks. We use the prefixes 'w', 'a', and 'f' to denote the input activations to the layer, output activations of self-attention and feedforward blocks respectively. Similarly we label each activation with an integer suffix corresponding to its token. Suffixes TP 1 and TP 2 denote the two tensor parallel partitions of the attention and feedforward blocks. The input batch consists of four tokens, with tokens 1 and 3 routed to the first expert (colored blue), and tokens 2 and 4 routed to the second expert (colored yellow).

Let us now discuss how our hybrid parallel algorithm computes the forward pass of an MoE

layer. As an example, we use an input batch with four tokens (numbered 1-4) in Figure 7.3. The tensor parallel group of GPUs 0 and 1 compute on tokens 1 and 2, whereas the tensor parallel group of GPUs 2 and 3 compute on tokens 3 and 4. Each GPU first computes their partition of the self-attention block (❶) and then issues an all-reduce (❷) to aggregate the complete output activations (prefixed by 'a') for their respective tokens. Now, each GPU applies the MoE routing function to their local tokens (❸). We assume that the routing function maps tokens 1 and 3 to the first expert i.e. feedforward 1, and tokens 2 and 4 to the second expert i.e. Feedforward 2. (❹) Now, an all-to-all communication primitive is issued in expert parallel groups to route the tokens as per the mapping decided by the routing function. Let us look at the expert parallel group of GPUs 0 and 2 to understand this all-to-all communication call. On GPU 0, token 1 has been mapped to the first expert and token 2 has been mapped to the second expert. Therefore, we want to retain a1 and send a2 to GPU 2 which houses the second expert. Similarly, on GPU 2, we want to retain a4 and send a3 over to GPU 0. Note that this communication pattern matches the semantics of an all-to-all communication primitive exactly. After the all-to-all has completed each GPU computes their tensor-parallel partitions of the expert feed forward blocks (❺) and issue an all-reduce to aggregate the complete output (❻). The final all-to-all communication call in the expert parallel groups (❼) essentially inverts the first all-to-all (❹) and brings back the tokens to their original GPUs. This is how our three dimensional hybrid parallel approach computes the forward pass of an MoE layer.

During the backward pass computation proceeds in the reverse direction i.e. (❼ - ❶). The all-to-all communication at ❼ and ❹ calls are reversed. For example, consider ❼, wherein the input to the all-to-all on GPU 0 would be gradients of the loss w.r.t. f1 and f2. Similarly for GPU 2, it would be the gradients w.r.t. f3 and f4. Now, after the all-to-all, the outputs on GPU 0

would be gradients of the loss w.r.t f1 and f3, and on GPU 2 it would be gradients of the loss w.r.t. f2 and f4. The all-reduce function calls (❹, ❻) are applied to the gradients w.r.t the input activations instead of the output. For more details about this all-reduce call, we refer the reader to Narayanan et al. [42]. Note that total amount of communication i.e. two all-reduces and two all-to-alls is the same as that of the forward pass. Finally, the data parallel groups synchronize their gradients via another all-reduce call, which completes the backward pass.

7.1.1 A Model for Memory Consumption

We now derive the extent to which TED can increase the base model sizes as compared to prior work like DeepSpeed-MoE [10], which only employ data and expert parallelism. Following previous work, we assume that every alternate layer has expert feedforward modules [101, 102, 104]. Let NP_{base} denote the number of parameters in the base model and E denote the number of experts. Let G be the number of GPUs. Note that two-thirds of the parameters in the base model reside in feed-forward blocks, and the remaining one-third in self-attention blocks [42]. Since only half of the feedforward blocks are designated as experts, the total number of expert parameters, NP_{exp}, in an MoE model are:

$$NP_{exp} = E \times \frac{1}{2} \times \left(\frac{2}{3} \times NP_{base} \right) = \frac{E}{3} \times NP_{base} \tag{7.2}$$

Now, the non-expert parameters are comprised of parameters in all the self-attention blocks and half of the feed-forward blocks. Thus, the total number of non-expert parameters, NP_{nonexp}, is

$$NP_{nonexp} = \frac{1}{2} \times \left(\frac{2}{3} \times NP_{base} \right) + \frac{1}{3} \times NP_{base} = \frac{2}{3} \times NP_{base} \tag{7.3}$$

Rajbhandari et al. [23] prove that the lower bound of memory consumption per GPU with ZeRO stage-1 is $\left(4 + \frac{12}{G_{data}}\right) \times NP_{gpu}$, where G_{data} is the degree of data parallelism and NP_{gpu} is the number of parameters of the model per GPU. Now, we use this formulation to derive a lower bound on memory consumption per GPU for TED as follows:

$$M_{gpu} \geq \left(4 + \frac{12}{G_{data}^{nonexp}}\right) \times NP_{gpu}^{nonexp} + \left(4 + \frac{12}{G_{data}^{exp}}\right) \times NP_{gpu}^{exp} \tag{7.4}$$

Here, NP_{gpu}^{nonexp} and NP_{gpu}^{exp} are the number of expert and non expert parameters per GPU. As discussed previously, G_{data}^{nonexp} and G_{data}^{exp} are the degrees of data parallelism for the non-expert and expert blocks respectively. Now, let us try to derive the values of NP_{gpu}^{nonexp} and NP_{gpu}^{exp}, starting with the former. MegatronLM's tensor parallelism divides the parameters of a model equally among the GPUs in a tensor parallel group. Since the size of a tensor parallel group in TED is G_{tensor}, we can write $NP_{gpu}^{nonexp} = \frac{NP_{nonexp}}{G_{tensor}}$. However, the expert parameters are divided within both the tensor parallel and expert parallel groups. As discussed previously, we use a degree of expert parallelism equal to the number of experts i.e. $G_{expert} = E$. Thus, $NP_{gpu}^{exp} = \frac{NP_{exp}}{G_{tensor} \times E}$. Also, it follows from Equation 7.1 that $G_{data}^{nonexp} = \frac{G}{G_{tensor}}$ and $G_{data}^{exp} = \frac{G}{G_{tensor} \times G_{expert}} = \frac{G}{G_{tensor} \times E}$. Substituting these values in Equation 7.4, we get

$$\begin{aligned} M_{gpu} &\geq \left(4 + \frac{12\,G_{tensor}}{G}\right) \times \frac{NP_{nonexp}}{G_{tensor}} + \left(4 + \frac{12\,G_{tensor}E}{G}\right) \times \frac{NP_{exp}}{G_{tensor}E} \\ &\geq \frac{4}{G_{tensor}}\left(NP_{nonexp} + \frac{NP_{exp}}{E}\right) + \frac{12}{G}(NP_{nonexp} + NP_{exp}) \end{aligned}$$

Now substituting from Equation 7.2 and 7.3, we get

$$
\begin{aligned}
M_{gpu} &\geq \frac{4}{G_{tensor}} \left(\frac{2}{3} NP_{base} + \frac{NP_{base}}{3} \right) + \frac{12}{G} (\frac{2}{3} NP_{base} + \frac{E}{3} NP_{base}) \\
&\geq \frac{4NP_{base}}{G_{tensor}} + \frac{4(E+2)}{G} NP_{base} \\
&\geq 4NP_{base} \times \left(\frac{1}{G_{tensor}} + \frac{E+2}{G} \right)
\end{aligned} \tag{7.5}
$$

Equation 7.5 can be used to derive an upper bound on the largest possible base model size that our framework can train, given enough number of GPUs. Note that as we increase the number of GPUs involved in training, the second term becomes negligible compared to the first. This gives us,

$$
\begin{aligned}
M_{gpu} &\geq \frac{4NP_{base}}{G_{tensor}} \\
\implies NP_{base} &\leq \frac{G_{tensor}}{4} \times M_{gpu}
\end{aligned} \tag{7.6}
$$

Note that substituting $G_{tensor} = 1$ in Equation 7.6 gives us the base model upper bound for Rajbhandari et al. [10], the current state-of-the-art for training MoEs. Thus, we have shown that our system enables the training of $G_{tensor}\times$ larger base models compared to the previous state-of-the-art. Note that the maximum degree of tensor parallelism is limited to the number of GPUs in a node due to performance considerations [42]. Nevertheless, our framework can still support $4\times$, $6\times$ and $8\times$ larger base models on Perlmutter, Summit and an NVIDIA-DGX-A100 machine respectively.

7.2 Memory Savings via Tiling

In the previous section, we provided an overview of how TED distributes the parameters and the computation of the forward and backward passes across the GPUs. However, a naive combination of tensor, expert, and data parallelism leads to significant spikes in memory usage during the optimizer step. Interestingly, the magnitude of this spike becomes worse as we increase the number of experts and/or the base model sizes. Note that it is important to resolve this issue so that we are able to fit MoEs with large base models in memory. Below, we discuss this phenomenon in detail and outline our solution to resolve this issue.

To demonstrate the aforementioned memory usage spike, we profile the memory consumed per GPU during various phases of training (forward pass, backward pass, optimizer step) for an MoE model with a 2.7B parameter base model and 32 experts, and show the results in Figure 7.4. We run this experiment on 32 GPUs of an NVIDIA DGX-A100 cluster with eight GPUs per node. We set the degree of tensor and expert parallelism to 1 and 32 respectively. This results in degrees of data parallelism as 32 and 1 for the non-expert and expert blocks respectively. We observe that memory consumption peaks during the optimizer step with a very significant spike of around 4.5 GB. An intermediate step in the optimizer phase in mixed precision training is the up-casting of 16-bit gradients to 32-bit gradients before the optimizer updates the weights. This requires the creation of a temporary buffer to store the 32-bit gradients and is exactly the reason why there is a significant increase in memory consumption. In fact, this problem becomes worse with increasing base model sizes and/or expert counts. Let us now understand why.

TED uses ZeRO stage-1 which reduces memory consumption by sharding the optimizer states and computation across the data parallel groups. Greater the degree of data parallelism, the

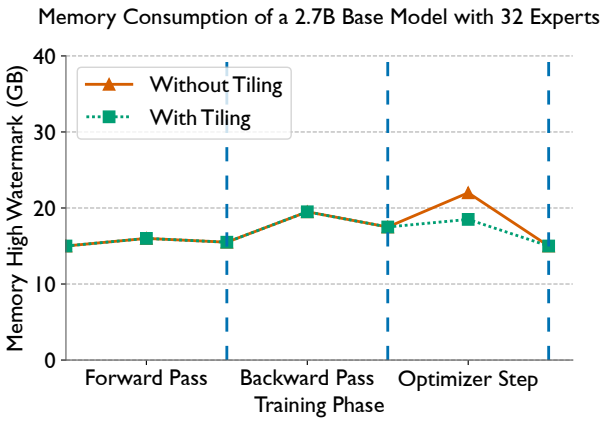

Figure 7.4: Memory consumption in the various phases of training for an MoE with a 2.7B parameter base model and 32 experts on 32 GPUs of an NVIDIA DGX-A100 (40 GB) cluster. We observe a large spike of an additional 4.5 GB in memory usage during the optimizer step (red), which is significantly reduced to around 1.5 GB by our tiled optimizer (green).

greater the reduction in memory consumption [23]. From the discussion in Section 7.1, we know that TED employs different degrees of data parallelism for the expert parameters and non-expert blocks. In fact, it follows from Equation 7.1 that

$$G_{tensor} \times G_{expert} \times G_{data}^{exp} = G_{tensor} \times G_{data}^{nonexp}$$

$$G_{expert} \times G_{data}^{exp} = G_{data}^{nonexp}$$

$$E \times G_{data}^{exp} = G_{data}^{nonexp}$$

$$G_{data}^{exp} = \frac{G_{data}^{nonexp}}{E} \tag{7.7}$$

From Equation 7.7, we can conclude that the degree of data parallelism for the expert blocks is $E\times$ less than that for the non-expert blocks. Therefore, ZeRO provides lesser memory savings

for the expert blocks than the non expert blocks. This is because the optimizer states for the expert blocks are sharded over $E\times$ lesser GPUs. Thus, as E increases each GPU has to process increasing number of parameters in the optimizer step. This leads to an increase in the size of the temporary 32-bit gradient buffer required to up-cast the expert parameter gradients. Increasing the base model size also worsens this problem as the size of the expert parameter group is directly proportional to the base model size. This is why it is imperative to resolve this issue such that we can train MoEs with large base models and/or large number of experts.

In this work, we propose a tiled formulation of the optimizer that strives to alleviate the aforementioned issue. Instead of processing the entire expert parameter group at once, we propose partitioning these parameters into "tiles" of a predefined size and iteratively processing these tiles. This ensures that at any given time, temporary 32-bit gradients are only produced for parameters belonging to a given tile. The temporary memory used to store these gradients can in fact be reused across tiles. For a tile size ts, we now only need $4 \times ts$ bytes of memory to materialize the 32-bit gradients. This makes the optimizer memory spike independent of the number of experts and the base model sizes! In our experiments, we fix the tile size to 1.8 million parameters, which essentially caps the spike in the optimizer step to 1 GB. We observed that this tile size is large enough to not cause any performance degradation due to the latency of multiple kernel launches. In Figure 7.4, we demonstrate how our tiled optimizer reduces the per GPU peak memory consumption for the aforementioned MoE with a 2.7B parameter base model and 32 experts by 3 GB. In fact, on another MoE with 6.7B parameters and 16 experts on 32 GPUs, our framework ran out of memory without tiling. Whereas, with tiling enabled, we were able to successfully train this model with a peak memory consumption of 31.3 GB. Since the maximum memory capacity of these GPUs is 40 GB, optimizer tiling provides a significant memory savings

of more than 21.75%!

7.3 Performance Optimizations

In the preceding sections, we focused on increasing the maximum possible size of MoEs that are supported by our framework. While the memory savings provided by expert and tensor parallelism contribute to this, they also result in a significant portion of the batch time being spent in expensive collective communication. In Figure 7.3, we can observe that the forward pass includes two all-reduce calls within the tensor parallel groups, and two all-to-all calls within the expert parallel groups. During the backward pass, these calls are repeated again. Also, large model training almost always uses activation checkpointing [77], which significantly reduces activation memory at the expense of a duplicate forward pass per layer. Thus, overall we end up with six all-to-all and six all-reduce communication calls, which become a significant bottleneck in training. We empirically demonstrate this in Figure 7.5 (leftmost bar titled Baseline), wherein we observe that almost half of the batch time is spent in the all-to-all and all-reduce communication calls. We will now describe two performance optimizations that seek to reduce the time spent in these communications and are extremely critical to the performance of our framework.

7.3.1 Duplicate Token Dropping (DTD) for Reducing Communication Volume

MegatronLM's tensor parallelism for partitioning self-attention and feed forward blocks involves issuing an all-reduce on local partial outputs to materialize the full outputs on each rank [39]. For example, in Figure 7.3, GPUs 0 and 1 issue an all-reduce (❷) after the self-attention block to assemble the full self-attention outputs for tokens 1 and 2. While, this leads

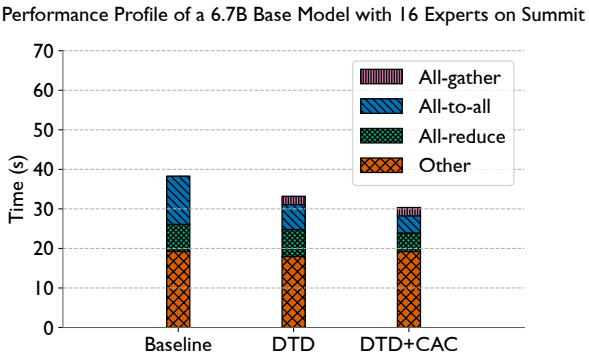

Figure 7.5: Impact of our communication optimizations on the batch time of an MoE model with a 6.7B parameter base model and 32 experts on 128 GPUs of Summit (batch size: 1024). Our optimizations result in significant reductions of 64.12% and 33% in the all-to-all and all-reduce time respectively, thereby improving the overall training time by 20.7%.

to duplication of activations across the tensor parallel ranks, it is not an issue for training regular transformer models (i.e. without experts) as the tensor parallel blocks under MegatronLM's algorithm require a complete set of input activations on each tensor parallel rank. Thus the duplicate activations output by a tensor parallel block serve as the required input for its successor. However, for MoEs, an unwanted side effect of this design choice is the presence of redundant tokens in the all-to-all communication calls. For example consider the first all-to-all in Figure 7.3 (❹). Self-attention output activations, a1 and a2, are communicated by both GPUs 0 and 1. Similarly, GPUs 2 and 3 both communicate a3 and a4. In general, the amount of unnecessary data in the all-to-all communication calls for a given token is proportional to the degree of tensor parallelism. Thus, naively combining expert and tensor parallelism can lead to the all-to-all communication becoming a significant bottleneck, especially as we try to increase the base model sizes (larger base models need more tensor parallelism). For example, in Figure 7.5, 32% of the batch time is spent in the all-to-all (leftmost bar titled baseline)! The degree of tensor parallelism and thus the degree of redundancy in the all-to-alls is four here.

To resolve this bottleneck, we propose duplicate token dropping (DTD), a communication

optimization geared towards eliminating unnecessary data in the all-to-all communication. We illustrate the working of DTD in Figure 7.6 for the first all-to-all communication in an MoE layer (❹ in Figure 7.3). Before the all-to-all is issued, GPUs within tensor parallel groups participate in a "drop" operation (❶ in Figure 7.6). The drop operation ensures that the there is no redundancy in the output activations across the tensor parallel ranks. For instance, GPU 0 drops the activation of a2 whereas GPU 1 drops the activation a1, thereby completely eliminating redundancy within their tensor parallel group. Similarly, GPUs 3 and 4 drop a3 and a4 respectively. The drop operation thus reduces the all-to-all message sizes by two times in this example, and in general the reduction is equal to the degree of tensor parallelism. However, after the all-to-all, the GPUs do not have the full input activations to commence the computation of the expert feed forward blocks. For instance, GPU 0 has the input activations for the token 1, but not for token 3 and vice versa for GPU 1. Therefore, to assemble the full input activations, we issue an all-gather call (❷ in Figure 7.6) between the tensor parallel GPUs. The all-gather ensures that the input dependencies for the expert feedforward blocks are met.

During the backward pass the all-gather call is replaced by a drop operation and the drop operation is replaced by an all-gather call. For the MoE model in Figure 7.5, we observe that DTD reduces the all-to-all communication time by 48%. While the inclusion of DTD leads to an additional all-gather operation (shown in red on top of the second bar), this overhead is outweighed by the improvement in the all-to-all communication timing. Overall, DTD results in an improvement of 13.21% in the batch time.

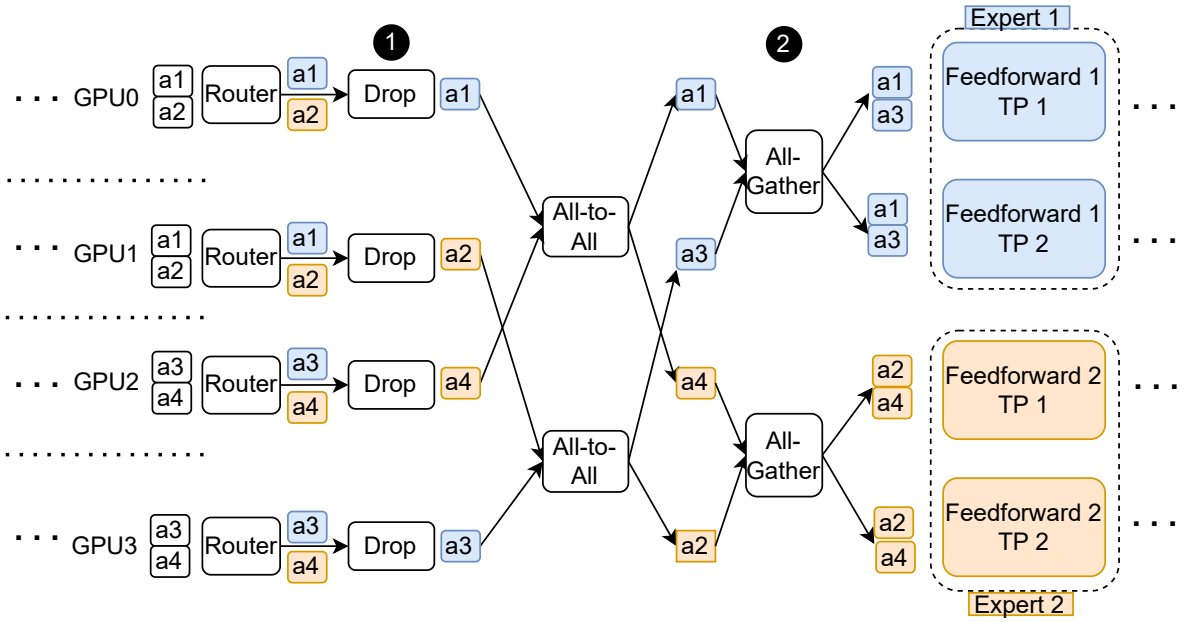

Figure 7.6: Duplicate token dropping (DTD) in the first all-to-all communication of an MoE layer (Steps 3–5 in Figure 7.3). Before the all-to-all, we apply the drop operation, which eliminates redundant tokens across tensor parallel ranks, and reduces the all-to-all message sizes by the degree of tensor parallelism. After the all-to-all, GPUs reassemble the full input to the feed forward blocks by issuing an all-gather between the tensor parallel ranks.

7.3.2 Communication-aware Activation Checkpointing (CAC)

We now turn our attention to a second source of redundant communication in large model training, namely activation checkpointing [77]. Intermediate activations in a neural network generated during the forward pass need to be stashed in memory as they are required during the backward pass for the gradient computation. However, for large model training, storing all the activations can lead to tremendous memory overhead. Activation checkpointing alleviates this issue by storing only a subset of the activations, which are essentially just the input activations of every layer. During the backward pass of a layer, the remaining activations are re-materialized from its stashed input activation by doing a local forward pass for that layer. Thus, activation checkpointing saves activation memory at the expense of a duplicate forward pass for every layer,

and is almost always used for training large neural networks. For more details, we refer the reader to Chen et al. [77].

We know from Section 7.1 that the forward pass of an MoE layer in TED involves two all-to-alls and two all-reduce calls in the forward pass and two all-to-alls and two all-reduce calls in the backward pass. Since activation checkpointing involves repeating the forward pass of a layer, we now end up with two additional all-to-all and all-reduce calls, thereby increasing communication volume by $1.5\times$ and making the training process inefficient.

To this end, we propose communication-aware checkpointing (CAC), a communication optimization that eliminates the additional communication in the second forward pass induced by activation checkpointing. During the first forward pass, CAC stashes the outputs of each all-reduce and all-to-all communication call along with the data stashed by standard activation checkpointing. Now, during the second forward pass, we bypass these communication calls and instead return the outputs for these communication calls stashed during the first forward pass. CAC thus reduces the communication volume by 33% at the expense of using extra GPU memory. For the MoE model in Figure 7.5, CAC indeed reduces the all-to-all and all-reduce communication times by 33% (compare second and third bars). In combination with DTD, the reductions in the all-to-all and all-reduce communication times are 64.12% and 33% respectively, amounting to a speedup of nearly 20.7% over the baseline version of DeepSpeed-TED.

7.4 Results

In this section, we discuss the results of the empirical experiments conducted to study the scalability of DeepSpeed-TED on the Summit supercomputer.

7.4.1 Validating Our Implementation

To verify the correctness of DeepSpeed-TED, we train an MoE with a 1.3B parameter base model and 4 experts on 8 GPUs of ThetaGPU and present the validation loss curve in Figure 7.7. We set $G_{tensor} = 2$, $_{expert} = 4$, $G_{data}^{nonexp} = 4$, and $G_{data}^{exp} = 1$. This allows us to test the correctness of our framework in a scenario where all three dimensions of its hybrid parallel approach are active. We also enable the communication optimizations discussed in Section 7.3 i.e. DTD and CAC. We observe that our framework is able to successfully train the model to convergence, and produces identical loss curves to DeepSpeed-MoE, a system that has been previously used to train state-of-the-art MoE models. In this way, we establish the correctness of our implementation.

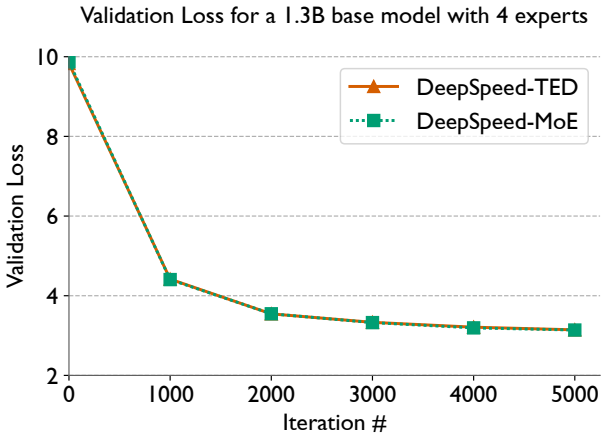

Figure 7.7: Validation loss for an MoE with a 1.3B base model and four experts on eight GPUs of ThetaGPU on the BookCorpus dataset [6]. We use a batch size of 128 and sequence length of 2048. We set $G_{tensor} = 2$, $G_{expert} = 4$, $G_{data}^{exp} = 1$, $G_{data}^{nonexp} = 4$.

7.4.2 Comparison of Supported Model Sizes

Figure 7.9 illustrates the results of our experiment in which we benchmark the largest MoE models that our framework and DeepSpeed-MoE [10] can train without running out of memory

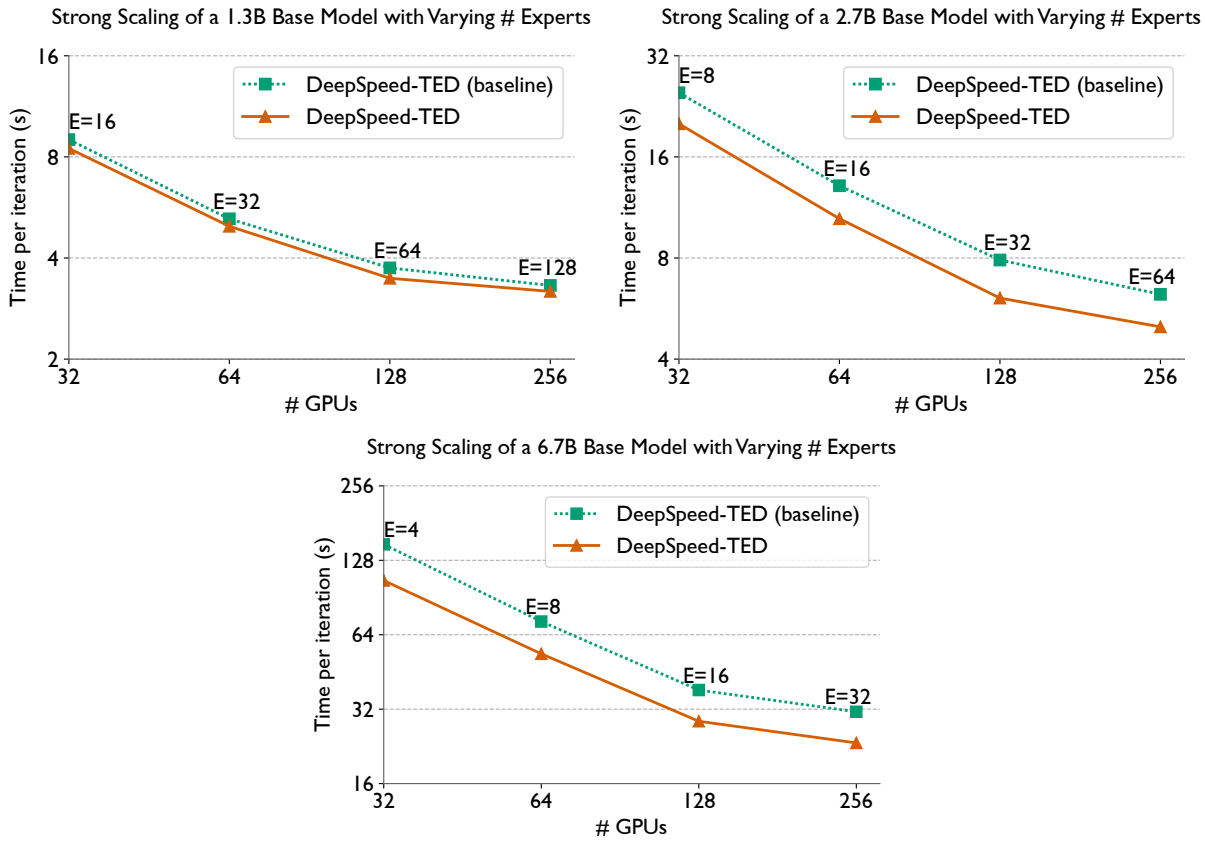

Figure 7.8: Strong scaling (with varying number of experts) of MoEs with the 1.3B, 2.7B and 6.7B parameter models in Table 4.1 used as base, on V100 GPUs of Summit. We annotate the plot with the number of experts used at each GPU count. We sample input batches of sizes 512, 512 and 1024 respectively, from the Pile dataset [9].

for various GPU counts ranging between 32 and 512. We use the base models in Table 4.1. To make sure that the experiment is fair to both the frameworks we do two things. While our proposed approach can in theory support arbitrarily large base models by increasing the degree of tensor parallelism, it is well known that tensor parallelism is extremely inefficient when used across nodes. Therefore, we only allow our framework to use a maximum tensor parallel degree of six, which is the number of GPUs on a node of Summit. Second, we limit the largest possible number of experts to 128 as prior work has demonstrated limited improvements in the statistical efficiency of a model beyond this number [101].

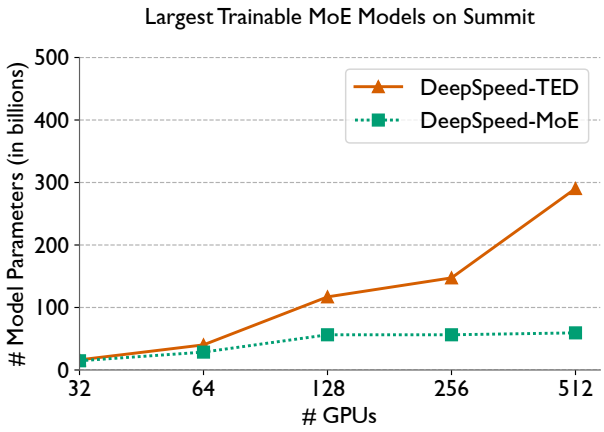

Figure 7.9: Largest MoE model sizes supported on various GPU counts on Summit. We construct MoEs using base models from Table 4.1 and number of experts in the range of 4 to 128. Compared to DeepSpeed-MoE [10], our framework supports 1.09-4.8$\times$ larger MoE models, with the ratio increasing with increasing number of GPUs.

Across the range of GPUs used in this experiment, we observe that DeepSpeed-TED supports $1.09 - 4.8\times$ larger MoE models than DeepSpeed-MoE. We also observe that this ratio increases as we increase the number of GPUs. This can be explained by Equation 7.5, which states that the memory consumption of our approach decreases with increasing number of GPUs. We observe that beyond 128 GPUs, our proposed framework can train MoEs with hundreds of billion of parameters on Summit, which is not possible with DeepSpeed-MoE. Thus, we have empirically demonstrated how our DeepSpeed-TED can enable the development of high quality MoE models, the parameters of which have been scaled along the base model dimension as well as the expert dimension.

7.4.3 Strong Scaling Performance

We now discuss the results of our strong scaling experiments, starting with the runs that varied the number of experts proportional to the number of GPUs. We demonstrate the results for the 1.3B, 2.7B and 6.7B base models in Figure 7.8. To demonstrate the efficacy of the commu-

nication optimizations discussed in Section 7.3, we also benchmark the baseline version of our framework i.e. with DTD+CAC disabled, and call it DeepSpeed-TED (baseline). Across all the figures, we observe that augmenting the training procedure with DTD and CAC indeed improves the hardware efficiency of training. However, while the speedups for the 2.7B and 6.7B parameter base models are significant: 19 to 23% and 25 to 29% respectively, our communication optimizations seem to be less effective for the smallest 1.3B base model providing modest speedups of around 4 to 7%. This is because at the given GPU counts and number of experts, ZeRO's memory optimizations and expert parallelism are able to fit this model in memory without the aid of tensor parallelism. Without tensor parallelism there is no redundancy in the all-to-all communication (see Section 7.3.1) and thus the DTD communication optimization is of no use in this scenario. Similarly, without tensor parallelism there is no all-reduce communication (❷ and ❻ of Figure 7.3). Thus, CAC only eliminates the unnecessary all-to-all calls, and is only partially applicable to this scenario. This explains the reduced effectiveness of our optimizations for the 1.3B base model.

Unlike the 1.3B base model, the 2.7B and 6.7B model require a tensor parallel degree of 2 and 4 to fit in available GPU memory. The ensuing redundancy in the all-to-all messages and the introduction of tensor parallelism thus makes our communication optimizations quite effective. Again, we observe larger speedups for the MoEs using the 6.7B base model (25–29% versus 19–23%) as a higher degree of tensor parallelism implies more redundancy in the all-to-all messages, which our optimizations successfully eliminate. It also implies a larger proportion of time spent in tensor parallel all-reduces which is significantly reduced by CAC.

In our strong scaling runs with fixed number of experts, we observed very similar absolute times per iteration and relative speedups for all the three models. For brevity, we only include

the results for the 6.7B parameter base model, and illustrate them in Figure 7.10. We have thus verified that our optimizations are effective at improving performance in two strong scaling setups across various base model sizes.

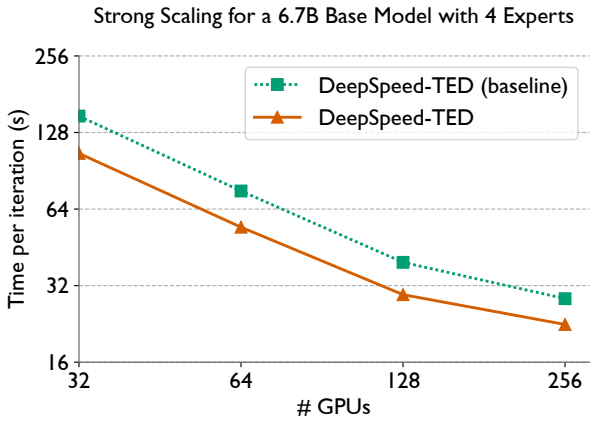

Figure 7.10: Strong scaling (with number of experts fixed to four) of a MoE with a 6.7B parameter model in Table 4.1 used as base, on V100 GPUs of Summit. We sample input batches of size 1024 from the Pile dataset [9].

7.4.4 Weak Scaling Performance

We also conduct a weak scaling experiment by fixing the number of experts to 16 and varying the base model size in proportion with the number of GPUs. We demonstrate the time per iteration (or batch) and percentage of peak half-precision throughputs for this experiment in Figure 7.11 and Table 7.1 respectively. Again, we observe a minor speedup of 6% for the 1.3B base model, and significant speedups of 20%, 25% and 36% for the 2.7B, 6.7B and 13B base models respectively. Just like the previous section, the progressively increasing effectiveness of our communication optimizations for larger base models can be explained by the correspondingly increasing degrees of tensor parallelism - 1, 2, 4, and 8. This creates more redundancy for the larger models in the all-to-all and increases the net communication volume of the all-reduces.

Note that while the speedups for the 13B parameter model is significant (36%), the hardware utilization for this model is extremely low. Even with our optimizations, we are only able to achieve 11.7% of the peak half-precision flop/s, which is significantly lower than the 1.3B (37% of peak), 2.7B (30% of peak) and 6.7B (27% of peak) base models. The explanation for this observation is that a tensor parallel degree of 8 for this model is greater than the number of GPUs on a Summit node. This experiment corroborates prior work which has observed that Megatron-LM's algorithm does not scale well beyond the confines of a node [42, 56].

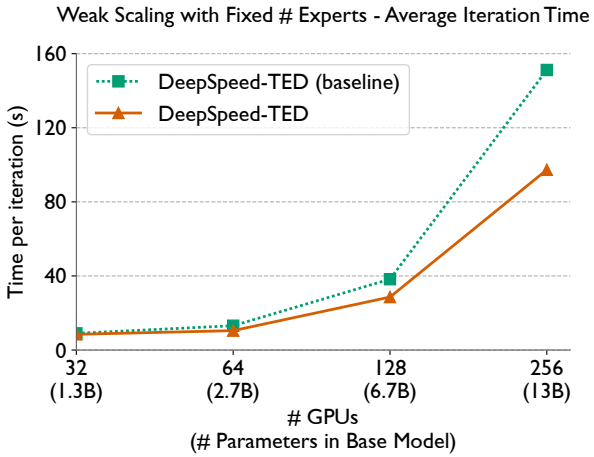

Figure 7.11: Average time per iteration (left) for a weak scaling study of MoE models with 16 experts on Summit. Base models and batch sizes are taken from Table 4.1.

Table 7.1: Percentage of peak half precision throughput for a weak scaling study of MoE models with 16 experts on Summit. Base models and batch sizes are taken from Table 4.1.

# GPUs	Base Model Size (# Parameters)	Throughput (% of peak)
32	1.3B	36.7
64	2.7B	30.0
128	6.7B	26.2
256	13.0B	11.7

Bibliography

[1] Tom B. Brown et al. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.

[2] Ohio State University. Osu micro-benchmarks 5.8. `http://mvapich.cse.ohio-state.edu/benchmarks/`.

[3] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. Sparse GPU kernels for deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020*, 2020.

[4] Haoran You, Chaojian Li, Pengfei Xu, Yonggan Fu, Yue Wang, Xiaohan Chen, Richard G. Baraniuk, Zhangyang Wang, and Yingyan Lin. Drawing early-bird tickets: Toward more efficient training of deep networks. In *International Conference on Learning Representations*, 2020.

[5] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *CoRR*, abs/1609.07843, 2016.

[6] Yukun Zhu, Ryan Kiros, Richard Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *arXiv preprint arXiv:1506.06724*, 2015.

[7] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development*, 39(5):575–582, 1995.

[8] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[9] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. The pile: An 800gb dataset of diverse text for language modeling. *CoRR*, abs/2101.00027, 2021.

[10] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale, 2022.

[11] Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. Reconciling modern machine-learning practice and the classical bias-variance trade-off. *Proceedings of the National Academy of Sciences*, 116(32):15849–15854, 2019.

[12] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020.

[13] Hugo Touvron et al. Llama 2: Open foundation and fine-tuned chat models. Technical report, 2023.

[14] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 24. Curran Associates, Inc., 2011.

[15] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 571–582, Broomfield, CO, October 2014. USENIX Association.

[16] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *NIPS*, 2012.

[17] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous sgd. In *International Conference on Learning Representations Workshop Track*, 2016.

[18] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 181–193, Santa Clara, CA, July 2017. USENIX Association.

[19] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: Experiences on accelerating data parallel training. *Proc. VLDB Endow.*, 13(12):3005–3018, August 2020.

[20] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow, 2018.

[21] Brian Van Essen, Hyojin Kim, Roger A. Pearce, Kofi Boakye, and Barry Chen. LBANN: livermore big artificial neural network HPC toolkit. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments, MLHPC 2015, Austin, Texas, USA, November 15, 2015*, pages 5:1–5:6. ACM, 2015.

[22] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[23] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[24] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[25] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. Pytorch fsdp: Experiences on scaling fully sharded data parallel. *Proc. VLDB Endow.*, 16(12):3848–3860, aug 2023.

[26] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. Vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49. IEEE Press, 2016.

[27] Mohamed Wahib, Haoyu Zhang, Truong Thao Nguyen, Aleksandr Drozd, Jens Domke, Lingqi Zhang, Ryousei Takano, and Satoshi Matsuoka. Scaling distributed deep learning workloads beyond the memory capacity with karma. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[28] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. ZeRO-Infinity: Breaking the gpu memory wall for extreme scale deep learning. SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[29] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017.

[30] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[31] Yang You, Igor Gitman, and Boris Ginsburg. Large batch training of convolutional networks, 2017.

[32] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. Imagenet training in minutes. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, New York, NY, USA, 2018. Association for Computing Machinery.

[33] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large batch optimization for deep learning: Training bert in 76 minutes, 2019.

[34] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus), 2020.

[35] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network, 2015.

[36] Yuxin Wu and Kaiming He. Group normalization, 2018.

[37] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.

[38] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France, 07–09 Jul 2015. PMLR.

[39] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. Technical report, 2020.

[40] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, Elton Zhang, Rewon Child, Reza Yazdani Aminabadi, Julie Bernauer, Xia Song, Mohammad Shoeybi, Yuxiong He, Michael Houston, Saurabh Tiwary, and Bryan Catanzaro. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. Technical report, 2022.

[41] BigScience. Bigscience large open-science open-access multilingual language model. `https://huggingface.co/bigscience/bloom`, 2022.

[42] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on GPU clusters. *CoRR*, abs/2104.04473, 2021.

[43] Qifan Xu, Shenggui Li, Chaoyu Gong, and Yang You. An efficient 2d method for training super-large deep learning models, 2021.

[44] Boxiang Wang, Qifan Xu, Zhengda Bian, and Yang You. Tesseract: Parallelize the tensor parallelism efficiently. In *Proceedings of the 51st International Conference on Parallel Processing*. ACM, aug 2022.

[45] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Sarikivi. Breaking the computation and communication abstraction barrier in distributed machine learning workloads, 2022.

[46] Nikoli Dryden, Naoya Maruyama, Tim Moon, Tom Benson, Marc Snir, and Brian Van Essen. Channel and filter parallelism for large-scale cnn training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[47] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, Sameer Kumar, Tongfei Guo, Yuanzhong Xu, and Zongwei Zhou. Overlap communication with dependent computation via decomposition in large deep learning models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS 2023, page 93–106, New York, NY, USA, 2022. Association for Computing Machinery.

[48] Zhiquan Lai, Shengwei Li, Xudong Tang, Keshi Ge, Weijie Liu, Yabo Duan, Linbo Qiao, and Dongsheng Li. Merak: An efficient distributed dnn training framework with automated 3d parallelism for giant foundation models. *IEEE Transactions on Parallel and Distributed Systems*, 34(5):1466–1478, 2023.

[49] Shengwei Li, Zhiquan Lai, Yanqi Hao, Weijie Liu, Keshi Ge, Xiaoge Deng, Dongsheng Li, and Kai Lu. Automated tensor model parallelism with overlapped communication for efficient foundation model training. *arXiv preprint arXiv:2305.16121*, 2023.

[50] Guanhua Wang, Chengming Zhang, Zheyu Shen, Ang Li, and Olatunji Ruwase. Domino: Eliminating communication in llm training via generic tensor slicing and overlapping, 2024.

[51] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. GPipe: efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[52] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. Regularized evolution for image classifier architecture search. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):4780–4789, Jul. 2019.

[53] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[54] Chiheon Kim, Heungsub Lee, Myungryong Jeong, Woonhyuk Baek, Boogeon Yoon, Ildoo Kim, Sungbin Lim, and Sungwoong Kim. torchgpipe: On-the-fly pipeline parallelism for training giant models. 2020.

[55] Arpan Jain, Ammar Ahmad Awan, Asmaa M. Aljuhani, Jahanzeb Maqbool Hashmi, Quentin G. Anthony, Hari Subramoni, Dhableswar K. Panda, Raghu Machiraju, and Anil Parwani. Gems: Gpu-enabled memory-aware model-parallelism system for distributed dnn training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[56] Siddharth Singh and Abhinav Bhatele. AxoNN: An asynchronous, message-driven parallel framework for extreme-scale deep learning. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, IPDPS '22. IEEE Computer Society, May 2022.

[57] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Granger, Phil Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *ACM Symposium on Operating Systems Principles (SOSP 2019)*, October 2019.

[58] Chi-Chung Chen, Chia-Lin Yang, and Hsiang-Yun Cheng. Efficient and robust parallel dnn training through model parallelism on multi-gpu platform, 2019.

[59] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural Networks*, 12(1):145–151, 1999.

[60] Rajeev Thakur and William D. Gropp. Improving the performance of collective operations in mpich. In Jack Dongarra, Domenico Laforenza, and Salvatore Orlando, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 257–267, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[61] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *J. Parallel Distrib. Comput.*, 69(2):117–124, feb 2009.

[62] Ernie Chan, Robert van de Geijn, William Gropp, and Rajeev Thakur. Collective communication on architectures that support simultaneous communication over multiple links. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, page 2–11, New York, NY, USA, 2006. Association for Computing Machinery.

[63] Richard L. Graham and Galen Shipman. Mpi support for multi-core architectures: Optimized shared memory collectives. In Alexey Lastovetsky, Tahar Kechadi, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 130–140, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[64] Krishna Kandalla, Hari Subramoni, Gopal Santhanaraman, Matthew Koop, and Dhabaleswar K. Panda. Designing multi-leader-based allgather algorithms for multi-core clusters. In *2009 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, 2009.

[65] Daniele De Sensi, Lorenzo Pichetti, Flavio Vella, Tiziano De Matteis, Zebin Ren, Luigi Fusco, Matteo Turisini, Daniele Cesarini, Kurt Lust, Animesh Trivedi, Duncan Roweth, Filippo Spiga, Salvatore Di Girolamo, and Torsten Hoefler. Exploring gpu-to-gpu communication: Insights into supercomputer interconnects. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '24. IEEE Press, 2024.

[66] Minsik Cho, Ulrich Finkler, David Kung, and Hillery Hunter. Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 241–251, 2019.

[67] Jahanzeb Maqbool Hashmi, Sourav Chakraborty, Mohammadreza Bayatpour, Hari Subramoni, and Dhabaleswar K. Panda. Designing efficient shared address space reduction collectives for multi-/many-cores. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1020–1029, 2018.

[68] Truong Thao Nguyen, Mohamed Wahib, and Ryousei Takano. Hierarchical Distributed-Memory Multi-Leader MPI-Allreduce for Deep Learning Workloads . In *2018 Sixth International Symposium on Computing and Networking Workshops (CANDARW)*, pages 216–222, Los Alamitos, CA, USA, November 2018. IEEE Computer Society.

[69] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. Synthesizing optimal collective algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, page 62–75, New York, NY, USA, 2021. Association for Computing Machinery.

[70] Sanghun Cho, Hyojun Son, and John Kim. Logical/physical topology-aware collective communication in deep learning training. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 56–68, 2023.

[71] Hao Feng, Boyuan Zhang, Fanjiang Ye, Min Si, Ching-Hsiang Chu, Jiannan Tian, Chunxing Yin, Summer Deng, Yuchen Hao, Pavan Balaji, Tong Geng, and Dingwen Tao. Accelerating communication in deep learning recommendation model training with dual-level adaptive lossy compression, 2024.

[72] Jiajun Huang, Sheng Di, Xiaodong Yu, Yujia Zhai, Jinyang Liu, Zizhe Jian, Xin Liang, Kai Zhao, Xiaoyi Lu, Zizhong Chen, Franck Cappello, Yanfei Guo, and Rajeev Thakur. hzccl: Accelerating collective communication with co-designed homomorphic compression. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '24. IEEE Press, 2024.

[73] Qinghua Zhou, Quentin Anthony, Lang Xu, Aamir Shafi, Mustafa Abduljabbar, Hari Subramoni, and Dhabaleswar K. DK Panda. Accelerating distributed deep learning training with compression assisted allgather and reduce-scatter communication, 2023.

[74] NVIDIA. Nccl. `https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/overview.html`.

[75] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. In *International Conference on Learning Representations*, 2018.

[76] Lisandro Dalcin and Yao-Lung L. Fang. mpi4py: Status update after 12 years of development. *Computing in Science Engineering*, 23(4):47–54, 2021.

[77] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *CoRR*, abs/1604.06174, 2016.

[78] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, page 300–314, New York, NY, USA, 2019. Association for Computing Machinery.

[79] Carl Yang, Aydin Buluc, and John D. Owens. Design principles for sparse matrix multiplication on the gpu, 2018.

[80] Siddharth Singh and Abhinav Bhatele. Exploiting sparsity in pruned neural networks to optimize large model training. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 245–255, Los Alamitos, CA, USA, may 2023. IEEE Computer Society.

[81] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks, 2016.

[82] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[83] Siddharth Singh, Prajwal Singhania, Aditya Ranjan, John Kirchenbauer, Jonas Geiping, Yuxin Wen, Neel Jain, Abhimanyu Hans, Manli Shu, Aditya Tomar, Tom Goldstein, and Abhinav Bhatele. Democratizing AI: Open-source scalable LLM training on GPU-based supercomputers. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '24, November 2024.

[84] Guanhua Wang, Heyang Qin, Sam Ade Jacobs, Xiaoxia Wu, Connor Holmes, Zhewei Yao, Samyam Rajbhandari, Olatunji Ruwase, Feng Yan, Lei Yang, and Yuxiong He. ZeRO++: Extremely efficient collective communication for large model training. In *The Twelfth International Conference on Learning Representations*, 2024.

[85] Rolf Rabenseifner. Optimization of collective reduction operations. In Marian Bubak, Geert Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *Computational Science - ICCS 2004*, pages 1–9, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[86] Edgar Solomonik, Abhinav Bhatele, and James Demmel. Improving communication performance in dense linear algebra via topology aware collectives. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11. ACM, November 2011. LLNL-CONF-491442.

[87] Abhinav Bhatele, Todd Gamblin, Steven H. Langer, Peer-Timo Bremer, Erik W. Draeger, Bernd Hamann, Katherine E. Isaacs, Aaditya G. Landge, Joshua A. Levine, Valerio Pascucci, Martin Schulz, and Charles H. Still. Mapping applications with collectives over sub-communicators on torus networks. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '12. IEEE Computer Society, November 2012. LLNL-CONF-556491.

[88] Ahmed Abdel-Gawad, Mithuna Thottethodi, and Abhinav Bhatele. RAHTM: Routing-algorithm aware hierarchical task mapping. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14. IEEE Computer Society, November 2014. LLNL-CONF-653568.

[89] Abhinav Bhatele, Nikhil Jain, Katherine E. Isaacs, Ronak Buch, Todd Gamblin, Steven H. Langer, and Laxmikant V. Kale. Optimizing the performance of parallel applications on a 5D torus via task mapping. In *Proceedings of IEEE International Conference on High Performance Computing*, HiPC '14. IEEE Computer Society, December 2014. LLNL-CONF-655465.

[90] Shaohuai Shi, Pengfei Xu, and Xiaowen Chu. Supervised learning based algorithm selection for deep neural networks. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 344–351, 2017.

[91] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. Technical report, 2019.

[92] Dhiraj D. Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhisek Kundu, Misha Smelyanskiy, Bharat Kaul, and Pradeep Dubey. A study of BFLOAT16 for deep learning training. *CoRR*, abs/1905.12322, 2019.

[93] Lightning AI. Litgpt. `https://github.com/Lightning-AI/litgpt`, 2023.

[94] Harsh Bhatia, Nikhil Jain, Abhinav Bhatele, Yarden Livnat, Jens Domke, Valerio Pascucci, and Peer-Timo Bremer. Interactive investigation of traffic congestion on fat-tree networks using TreeScope. *Computer Graphics Forum*, 37(3):561–572, June 2018.

[95] Misbah Mubarak, Philip Carns, Jonathan Jenkins, Jianping Li, Nikhil Jain, Shane Snyder, Robert B. Ross, Christopher D. Carothers, Abhinav Bhatele, and Kwan-Liu Ma. Quantifying I/O and communication traffic interference on dragonfly networks equipped with burst buffers. In *Proceedings of the IEEE Cluster Conference*, Cluster '17, September 2017. LLNL-CONF-731482.

[96] Junqi Yin, Sajal Dash, Feiyi Wang, and Mallikarjun Shankar. Forge: Pre-training open foundation models for science. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA, 2023. Association for Computing Machinery.

[97] Sajal Dash, Isaac R Lyngaas, Junqi Yin, Xiao Wang, Romain Egele, J. Austin Ellis, Matthias Maiterth, Guojing Cong, Feiyi Wang, and Prasanna Balaprakash. Optimizing distributed training on frontier for large language models. In *ISC High Performance 2024 Research Paper Proceedings (39th International Conference)*, pages 1–11, 2024.

[98] Hewlett Packard Enterprise. *HPE Cassini Performance Counters*, 2024. Accessed: 2025-02-24.

[99] Jonas Geiping, Sean McLeish, Neel Jain, John Kirchenbauer, Siddharth Singh, Brian R. Bartoldson, Bhavya Kailkhura, Abhinav Bhatele, and Tom Goldstein. Scaling up test-time compute with latent reasoning: A recurrent depth approach, 2025.

[100] Siddharth Singh, Olatunji Ruwase, Ammar Ahmad Awan, Samyam Rajbhandari, Yuxiong He, and Abhinav Bhatele. A hybrid tensor-expert-data parallelism approach to optimize mixture-of-experts training. In *Proceedings of the 37th International Conference on Supercomputing*, ICS '23, page 203–214, New York, NY, USA, 2023. Association for Computing Machinery.

[101] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity, 2021.

[102] Young Jin Kim, Ammar Ahmad Awan, Alexandre Muzio, Andres Felipe Cruz Salinas, Liyang Lu, Amr Hendy, Samyam Rajbhandari, Yuxiong He, and Hany Hassan Awadalla. Scalable and efficient moe training for multitask multilingual models, 2021.

[103] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. Zero-offload: Democratizing billion-scale model training. *CoRR*, abs/2101.06840, 2021.

[104] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding, 2020.