Towards More Efficient Communication for Distributed Learning Systems

Liang Luo

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2020

Reading Committee:

Luis Ceze, Chair

Arvind Krishnamurthy

Jacob Nelson

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Towards More Efficient Communication for Distributed Learning Systems

Liang Luo

Chair of the Supervisory Committee:
Professor Luis Ceze
Computer Science and Engineering

The explosion of data volume and ever-increasing speed of accelerators shift the bottleneck of large-scale distributed training tasks from computation to communication. We observe significant pressure on the communication backends of various mainstream learning systems in multiple environments when running such tasks. Achieving efficient large scale learning relies on more effective communication planes.

We provide detailed analysis that root-causes the bottlenecks affecting the communication efficiency of these systems in the context of different environments. We pinpoint such bottlenecks from the software, hardware and network infrastructure stacks.

We show how these obstacles can be overcome with a systematic codesign of a streamlined communication stack, a balanced hardware and cluster configuration with the distributed training workload, together with awareness of network topology and environment. We show this series of approaches, named Parameter Box, Parameter Hub, Parameter Link along with ☁Collectives, accelerate distributed training from small clusters to datacenters and all the way to the commercial clouds while providing varying degrees of customization to suit different needs.

DEDICATION

Thank you to my parents, Changyi and Yanan, for your endless love and support.

Thank you to my academic advisers, Luis Ceze and Arvind Krishnamurthy, for patiently guiding me through the process.

Thank you to this committee for keeping me on track.

Thank you to my friends for the joy and laughter in the journey.

Chapter 1

INTRODUCTION

Today, training systems have gained significant attention in the literature for the overwhelming popularity of machine-learning (ML) related workloads. Most of work to date in the system and architecture community has focused on improving the efficiency of evaluating trained models. This makes sense given that a model is trained only once but can be used many times for inference. However, arriving at a trained model frequently requires experimentation, and thus multiple training runs, each of which may take days. Accelerating the training process lets ML scientists iterate faster and design better model.

Traditionally, training has been viewed as a compute-bound problem, best done in a single large compute node with many accelerators. However, the ever-growing data volume pushes for monster-sized models, a scale even the exponentiation of compute power growth in single device couldn't handle. As ML models get bigger, training time gets prohibitively longer. Timely training requires exploiting parallelism with a distributed system.

Table 1.1 gives a taste of the evolution of machines learning models/techniques: they are getting increasingly more demanding in terms of compute resources, so much that the use of hundreds or even thousands of machines and accelerators for parallel training for weeks is commonplace.

The most common way of exploiting parallelism, "data" parallelism, consists of a computation-heavy local computation phase and a communication-heavy parameter exchange phase. Efficient distributed training involves optimizing both stages. Many past work demonstrate the solution of accelerating these processes by building specialized hardware clusters with fast interconnects [136, 157, 2, 68, 98, 140, 53, 62, 85]. While the results are encouraging, as we shall see, faster hardware is not panacea: the unstreamlined software stack and unbalanced

Models	Complexity	Size	Time	Configuration
Linear Regression [127]	O($D^3 + N^2D$)	Small	Short	CPU
SVM [148]	O(N^2D^2)	Small	Short	CPU
GBDT [48]	O($TLFN$)	Small	Short	A few CPUs
AlexNet [77]	0.0058	48M	6 days	2x GTX 580
ZFNet [158]	0.0062	≈48M	12 days	GTX 580
VGGNet [135]	0.12	137M	15 days	4x Titan Black
GoogleNet [143]	0.03	9M	7 days	A few GPUs
ResNet [58]	0.117	58M	21 days	8x GPUs
Xception [29]	5.0	22M	30 days	60x K80
BERT [38]	3.8	340M	4 days	16x TPUs v2
GPT-2 [112]	248	1.5B	7+ days [119]	256x TPUs v3
Turing-NLG [97]	Unknown	17B	Unknown	256x V100
NAS [162]	31	86M	28 days	800 K40 GPUs
AlphaGoZero [134]	1800	-	1 day	5000x TPUs

Table 1.1: A few representative machine learning techniques and models that support increasingly complex tasks (trees, support vector machines, neural networks) and their complexities to train (pfs-day, or days to train computing at 1 PFlop/s. 1PF/s roughly corresponds to 64 Tesla V100 GPUs FP32, or 8 with mixed-precision FP16, running at peak throughput), and their reported training time and the machines they are trained on. We used the same method of estimating complexity as in the original OpenAI blog post [111] for the additional models in the table. D: features. N: samples. L: leaves per tree. T: number of trees to build.

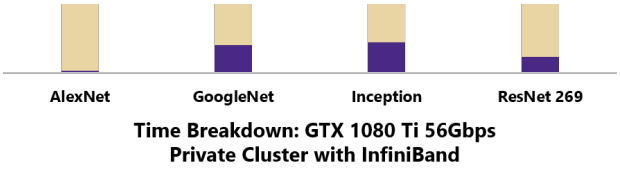

Figure 1.1: Even with a fast, small-scale InfiniBand cluster, communication still can take up to 90% of the time duration training, wasting compute resources.

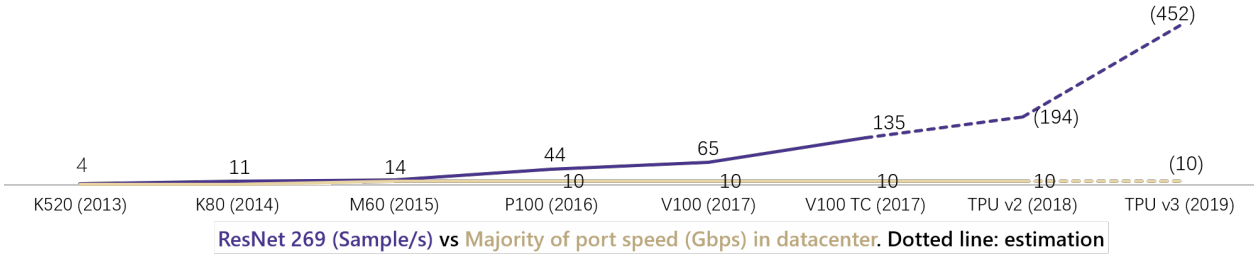

Figure 1.2: Throughput for an industry standard benchmark (ResNet) has seen an improvement of 35x, and is estimated to increase by 100x with latest accelerators. GPU performance tested with MxNet on EC2. TPU throughput estimated based on TensorCore's measured performance. Estimation for majority of the port speed of datacenters is based on Cisco study [30]. Only until very recently did the public cloud start offering 100Gbps bandwidth instances on standalone VMs [95, 9]

.

compute to communication resource allocation prohibits utilizing full capability of the hardware, and the compute units are more likely than not waiting on the network because the latter simply cannot keep up (Figure 1.1). We expect the problem not going away on its own, because of the observed trend at which the network capability is growing is significantly outpaced by that of the compute resources (Figure 1.2).

To tackle these problems, a comprehensive codesign of software stack and hardware configuration is required. To that end, we provide a template for an entity called Parameter Box that provides the near perfect communication to computation balance for acting as a parameter server in a distributed training job, and a companion piece of software, Parameter Hub that streamlines handling of gradient transfer, aggregation and model optimization by

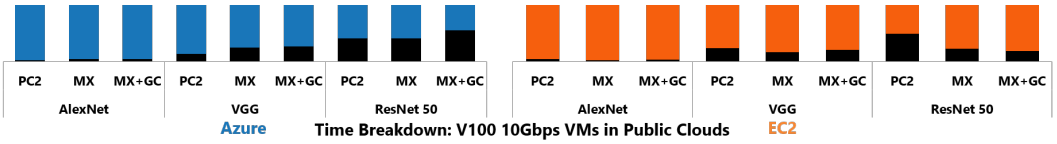

Figure 1.3: Even with state of the art training frameworks and recent optimizations, up to 90% of time during cloud-based training of popular models is wasted explicitly waiting on the network in the public clouds.

carefully extracting locality inside a physical host and latency hiding.

While Parameter Box and Parameter Hub are effective, it is important to understand accelerating training in these specialized, privately-owned clusters does not complete the story: first, those hardware setups require steep investment and only a few have that luxury; second, since it is much easier to thoroughly optimize the entire stack in private clusters as we have control over the entire system, some optimizations are not always possible to be applied universally.

An alternative to owning a private cluster is renting VMs from the public cloud, which has become a popular, more accessible approach. Currently, all major cloud providers offer racks of nodes with specialized accelerators (such as GPUs, TPUs, and custom FPGAs) [51, 96, 7, 6, 47, 70, 151] for ML workloads.

But at the same time, even with modern frameworks and recent optimizations (e.g., gradient compression and quantization [82, 128, 81], latency-hiding [161, 66, 57], optimized communication libraries [43, 110, 101] and large batch optimizations [53]), distributed training at scale on the public cloud still incurs high overhead: up to 90% of total training time can be wasted waiting on the network (Figure 1.3). Further, existing solutions so far focus solely on addressing the *bandwidth* bottleneck, and ignores cloud-specific challenges: the hierarchical network structure in datacenter breaks the usual assumption of link speed being uniform; multi-tenancy and the dynamic nature of the cloud traffic cause high variation in performance. All these add to the complexity of scaling up distributed training and can

render existing solutions less effective.

Accelerating cloud-based distributed training thus requires paying attention to a third dimension (apart from software and hardware): the environment. First, we need an hierarchical aggregation mechanism that is appropriate for network topologies that display bandwidth oversubscription, and that makes appropriate use of underprovisioned links. Second, we need to be able to identify the underlying network topologies and bandwidth/latency constraints (or collectively, locality) even if the public cloud does not expose such information. Finally, we need communication schemes that can react to changing network conditions, especially in the presence of interfering traffic generated by other tenants. Our solution is collectively called Parameter Link, an optimized, locality-aware system that uses a fitted hierarchical aggregation scheme to extract locality from the underlying datacenter network, based on end-to-end network probes and dynamic network load.

We extend Parameter Link to support collectives in order to benefit a wider range of existing systems that use collectives and run in a highly specialized networks that may not have a standard datacenter fat tree topology, through a fully-transparent rank reordering scheme. This extension, ☁Collectives, is non-intrusive, requires no code changes nor rebuilding of existing application.

Parameter Box, Parameter Hub, Parameter Link and ☁Collectives essentially represent optimizations from hardware, software and awareness of the network topology and environment, for parameter servers, hierarchical schemes and collectives. By breaking the optimizations into four broad categories, we provide a mix-and-match style of choices of optimizations for users to accelerate training tasks of all sizes, ubiquitously, to suit their specific needs.

In the following sections, we start with an overview of the mechanism of distributed training, then we walk through each of the proposed solution in detail. We show how specific optimizations adopted target directly at the bottlenecks in each scenario and lead to end-to-end speedup in real-world training tasks.

1.1 Background

We now establish conventions used in this paper, and familiarize the reader with basic concepts of distributed training.

1.1.1 Training a ML model

Different types of models may appear to have different ways of training, but sitting at the center of most models is the common notion of parameters and gradients. Parameters are the goals: they define the ultimate models; gradients are the means: they are derived with respect to parameters, and guide how parameters should be adjusted to minimize the errors (together with a learning rate), usually in an iterative manner, using the technique gradient descent (SGD) [21, 121] (or its variant).

To illustrate the training process, we use the example of training a deep learning (DL) model, or a deep neural network (DNN), for its popularity. We will continue to explain in the context of DL models throughout this paper for consistency. Modern DL models can have hundreds of *layers* making up multi-megabyte-size *models*. The training process has three phases. In the *forward pass*, a prediction is generated for an input. In the *backward pass*, the prediction is compared with a label to calculate prediction error; then, through *backpropagation* [122], the gradient for each parameter is calculated with respect to this error. The model is then *updated* using these gradients, often using a variant of the SGD algorithm. Each Computation is often done on GPUs or other accelerators suited to regular data-parallel operations, processing a batch of samples at once (*minibatching*).

1.1.2 Distributed Training

Broadly speaking, there are two extremes in terms of paradigms in distributed training: *data* parallelism and *model* parallelism, and many hybrid systems strike a balance between these two.

In data parallelism, training data is pre-partitioned to each individual workers, and each

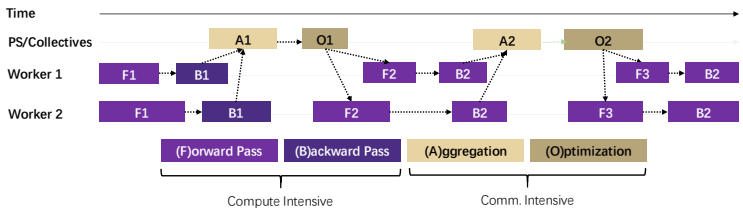

Figure 1.4: A few iterations of distributed training pipeline of neural networks.

worker sees only a partition of the data. In model parallelism, instead of partitioning training data, the model itself is being partitioned.

This paper mainly focuses on data parallelism, as it is the default choice in many frameworks and practices. The distributed training process (Figure 1.4) using data parallelism is different in a few ways from training within a single node. First, a mean gradient is calculated across all minibatches in each machine. Then, the mean of the gradients from each machine is calculated. Finally, the model is updated based on that mean, new parameters are broadcast to each machine and GPU, and the next batch is trained. This paper focuses on optimizing calculation of both the mean gradient across machines and the subsequent model update (or *parameter exchange*). Note that gradient aggregation and model optimization are both element-wise operations.

The process described here is *synchronous training*, where all machines and GPUs execute a new minibatch simultaneously and update the model based on the gradients in the current iteration. It is also possible to train asynchronously [1, 25, 32, 118, 27, 37], sacrificing reproducibility for a potential throughput increase. We focus on synchronous training due to its simplicity and commonality in industry, but our techniques can also benefit asynchronous training.

1.1.3 Datacenter Network Topology

A typical datacenter network has a hierarchical, multi-tiered topology [103, 54, 120, 83] (Figure 1.6). Machines are grouped into *racks*, each connecting to a top-of-rack *(ToR)* switch.

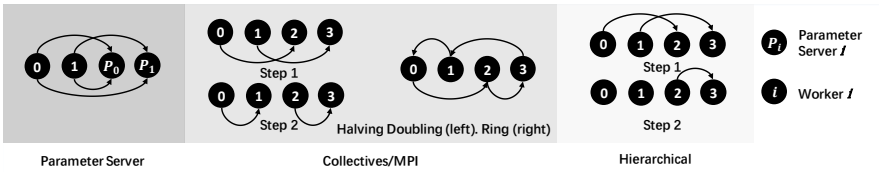

Figure 1.5: Aggregation is commonly done with one of the three prevailing paradigms, parameter server, collectives all-reduce, and hierarchical aggregation in practice.

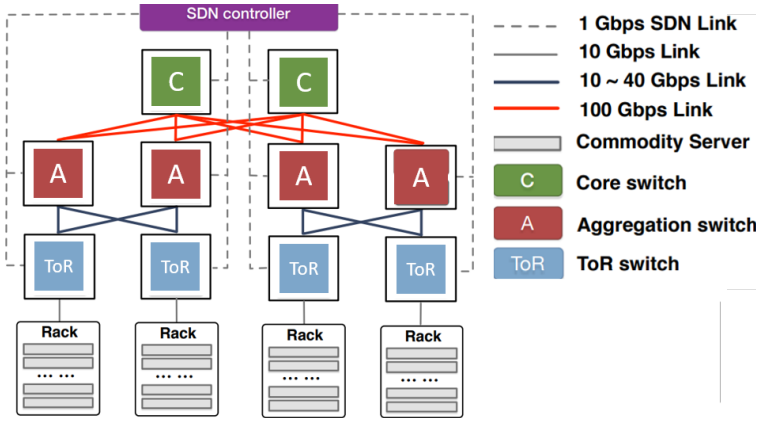

Figure 1.6: Overview of a typical datacenter network topology.

ToR switches are connected to multiple upper level devices. This setup poses challenges for existing training systems: in this setting, the communication performance of two end-hosts is affected by where they reside: they enjoy full link bisection bandwidth within a rack, because link capacity is not shared at the rack-level, but if they are on different racks, the communication performance depends on link congestion and oversubscription ratio [19]. In this paper, we use *locality* to refer to the cause of variation in communication performance, which includes: (1) physical topology: the location of the nodes and (2) dynamic network load. Efficient communication requires carefully architecting software to tap into both aspects [67, 41]: solutions that ignore physical topology are subject to long-term communication imbalances, and those that ignore dynamic network load suffer from short-term inefficiencies.

1.1.4 Distributed Training is Here to Stay

Faster and more powerful accelerators (most notably TPUs [70] and Nvidia DGX [105]) open up the possibility of training in a single device. It all of sudden seems plausible to build a "supercomputer" for training, which completely eliminates the need for costly communication.

Unfortunately, building a training supercomputer does not avoid the problem of insufficient compute resources, but only delays it, for at least two reasons: from a historic perspective, it never happens that there is a surplus in the compute power when it comes to new models, as shown in Table 1.1: scientists can always find models whose complexities are well beyond reach of a single device, which frequently ended up being trained in a distributed fashion: i.e., the use of a single device (e.g. DGX) is not because that single device covers all the need, but rather the lack of additional devices; from an architecture perspective, compute density cannot scale forever as many hard limits are imposed on the number of transistors to fit on a fixed area, including physical effects and cooling constraints. When near these limits, it gets prohibitively difficult to build faster chips within a confined area.

On the other hand, any training that spans device boundary, not necessarily machine boundary, can be classified as distributed training, and the communication medium need not be limited to conventional network media, but can also include device buses. With this broad view, distributed training is inherent in deep learning. In fact, many have already started looking into optimization of inter-device communication within a single machine [146].

1.1.5 Gradient Aggregation: Common Practices

In a loose taxonomy, collecting gradients for aggregation (known as parameter exchange) is commonly done with one of the following paradigms (Figure 1.5).

Parameter Servers (PS) [137, 79, 80, 160, 85, 86, 161, 33]. PSs are key-value stores, where keys and values represent the model's layer IDs and parameters. PSs are well-suited for training at a small scale. PSs can be centralized or sharded. In each iteration, all workers update the model stored in PSs with their locally-produced gradients. PS configurations

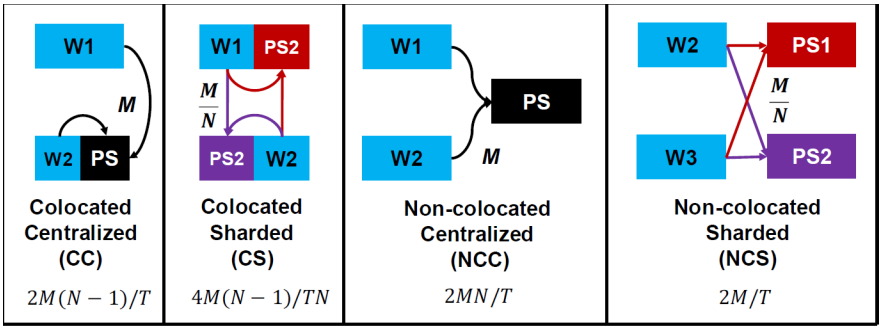

Figure 1.7: Aggregation is commonly done with one of the three prevailing paradigms, parameter server, collectives all-reduce, and hierarchical aggregation in practice.

primarily differ along two axes: colocated (C) versus non-colocated (NC), and centralized (C) versus sharded (S). A PS setup is colocated if a worker and a server process share the same physical machine. A PS setup is centralized if a single PS process handles all keys; and a sharded setup load-balances keys across multiple PS processes. During synchronization, each worker sends and receives model updates from each PS process. Figure 1.7 illustrates the four combinations of choices from these two axes: Colocated Centralized (CC), Colocated Sharded (CS), Non-colocated Centralized (NC) and Non-colocated Sharded (NCS).

In general, sharded PSs scale better at higher hardware costs. Colocated PSs reduce total data movement on the network by $\frac{1}{N}$ with N workers participating: the update for the partition of the model assigned to a colocated PS need not go through the network. While many frameworks default to CS configurations [102, 42], in a colocated setup the PS process interferes with the training process, because both are contending for network and processing resources. Specifically, compared to NC PSs, *each network interface must process roughly 2x the network traffic, because both the colocated worker and PS processes must send and receive model updates from remote hosts*, creating a major bottleneck in network-bound distributed training.

Collective AllReduce (CA) [123, 144, 117, 20, 17]. Popular in the context of MPI, CAs

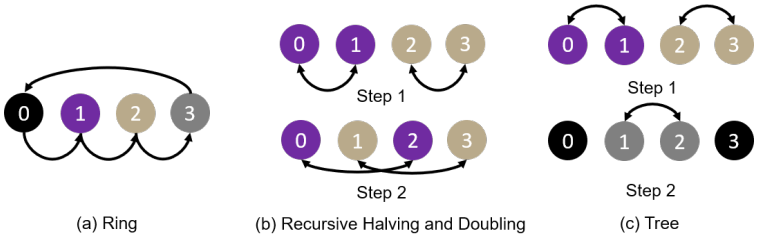

Figure 1.8: Rounds of communications (color-coded) in various popular collectives algorithms with 4 nodes.

are widely used in larger-scale training. All nodes in CA participate in the communication, usually running symmetric tasks. The end goal of CA is that all nodes have a globally-reduced copy of the data. Widely used CA in training deep learning models include halving-doubling [53], ring and double binary tree [110, 129]. We characterize some of the popular ☁Collectives here.

Ring [113]. As shown in Figure 1.8(a), ring algorithm works by connecting nodes to form a virtual ring. Data is then passed along the ring sequentially. Ring algorithm requires $O(N)$ steps to complete, sending $O(NS)$ amount of data.

Halving Doubling [144]. As shown in Figure 1.8(b), halving doubling works by recursively doubling the distance (in terms of rank ID) while halving the total amount of data sent in each round, requiring $O(log_2 N)$ steps to finish while also sending $O(NS)$ amount of data on wire.

Tree. Tree algorithms are versatile, in a simple form a single tree is built where data is transferred from leaves to the root and vice-versa [62]; in a more optimized setting, a pair of complementary binary trees are built to fully utilize the full bisection bandwidth [124], each sending and receiving $S/2$. For binary tree collectives algorithms, $O(log_2 N)$ rounds of communication are required, also sending $O(NS)$ bytes on wire.

BCube [44]. BCube is very similar to halving doubling from a structural perspective, in the sense that nodes are organized into group of B peers. BCube operates in $O(log_B N)$ rounds, and each node in each round would peer with an unique node in another $B-1$ groups. Each

node communicates $\frac{S}{B^i}$ amount of data in round i. BCube achieving a total bytes on wire of $O(\sum_{i=0}^{log_B N-1} \frac{S}{B^{i+1}})$.

CAs and PSs are not mutually exclusive. Using one does not exclude the use of other. For example, [74] dynamically chooses CA or PS based on sparsity of a given key.

Gradient aggregation can be *flat* (e.g., use of PSes and CAs are generally flat) or *hierarchical* (Hierarchical Aggregation, HA), which refers to the generic technique of aggregating data in multiple steps, from local to global. Exemplar usage of HA in the distributed training context include [62, 28, 49, 99], though in the context of proprietary networks. HA is highly flexible and supports mix and matching of multiple aggregation paradigms [50, 3]. Hierarchical schemes are essential in enabling large-scale training.

This paper mainly focuses on the discussion of building efficient PSs, but most of the techniques still apply to accelerating CAs as well. We dedicate sections related to ☁Collectives to CAs.

1.1.6 More Efficient Distributed Training: Prior Arts

Approaches to accelerating distributed training can be classified into one of the broad categories in the current literature.

Synchronize less often. One way to achieve a lower synchronization frequency is to oversubscribe GPUs. This can be done by using a very large batch size, fully utilizing GPU memories, making GPU compute the bottleneck [46, 53, 139, 68, 155, 154]. Large batch sizes reduce communication frequency. However, this eliminates the potential of achieving a larger speedup with fast communication. For example, with ResNet-50, [131] shows only 10 samples are needed to fully utilize a recent GPU. This means the computation of large batches can be further spread to more GPUs, and additional throughput is attainable provided that communication overhead is low. Further, large batch optimization is also not universally available (requiring GPUs with large memory) and may be subject to worse generalization [73].

Orthogonal to large batch optimization, another line of work target at less synchro-

nization. They tune the consistency model of distributed training, with relaxed consistency [35, 59, 31, 149, 116, 152, 147]. Generally, these relaxed consistency models do not mandate a strict barrier for synchronization at iteration boundary, and instead, they allow staleness in the model and a potential different view of model from each worker, removing the synchronization overhead from the critical path. However, these methods suffer from difficulty in reproducing the models.

Send less data. Sending less data accelerates distributed training in a bandwidth-bound environment, and can be achieved through (1) lossless compression [24]; (2) lossy compression, removing redundancy in the SGD algorithm [82]; (3) quantizing update gradients to low bit representations and locally apply residual errors [128, 81] and (4) decomposing large update matrices [27, 161, 153] and reconstructing at destination. These methods either trade more computation for less communication, or risk affecting the final convergence accuracy of the model, and both of which may turn out *increasing* the total wall clock time required to reach the target accuracy.

Build faster clusters. Another series of work involves building specialized hardware clusters for distributed training with quick interconnects to tackle communication bottlenecks [136, 157, 2, 68, 98, 140, 53, 62]. While the results have been encouraging, these approaches demand steep investments and are not available to everyone.

Hide communication latency. Most modern frameworks encode the model being trained as a dataflow graph. An operator is executed as soon as its dependencies are resolved, and this allows overlapping of communication and computation during the backpropagation stage of distributed training. Some work even attempt to re-prioritize sending of first layers over later layers to deal with this priority inversion problem. Notable applications of this idea include [57, 13, 161, 114]. However, communication latency hiding has severe limits: faster computation device leaves smaller room; many model training is in fact bandwidth bound, and hiding latency only has limited impact on the total training time.

Use finer-grained parallelism. This can be done by blending in higher compute hardware utilization of data parallelism and lower communication cost of model parallelism to form

pipelined parallelism [56]. This can also be done by allowing a flexible combination of slicing along arbitrary dimensions of S(ample)O(perator)A(ttribute)P(arameter), enabling more execution possibilities and effectively enlarging the action space. More efficient schedule can then be determined by intelligently searching through the enlarged space by taking communication cost into account [69]. However, these methods have the limit of needing to research as soon as the underlying hardware environment, or the models being trained change.

Accelerate at network level. The emergence of programmable network devices open up the opportunity to accelerate distributed training at the core of network, allowing gradient aggregation in the network devices, resulting in lower parameter exchange latency and lower bandwidth requirement (e.g. broadcast of aggregated model can be efficiently done with a network switch [125, 84]). Current programmable devices are not without limits, enforcing hard constraints on compute and memory.

Improving cluster-level efficiency. A series of work [151, 131] target at an orthogonal goal, and instead of optimizing for each individual task, they aim to achieve an average high utilization of compute resources in a cluster, by time-sharing (preemption) and better placement.

Chapter 2

PARAMETER BOX: A BALANCED HARDWARE DESIGN FOR PARAMETER SERVERS AT RACK-SCALE

We now describe Parameter Box, the solution to inefficient distributed training for clusters with a flat network topology (rack-level). Parameter Box provides a re-design of a PS at the hardware level, and mostly applies to situations where the user assumes full control over the cluster. We start with the current problem with the PS software and hardware that runs it. When deployed, Parameter Box serves as a centeralized parameter server in the cluster.

2.1 Insufficient Bandwidth and Overprovisioned Compute Resources in Rack-Scale Clusters

Centralized PSs have lower cost than NCS PSs, and half of the bandwidth stress compared to CS PSs on each interface card. Thus it is desirable to have a centralized reduction entity at rack level. However, scaling a centralized PS to rack scale is challenging [62]. The root cause is hardware imbalance in the allocation of computation and communication resources in the host: centralized PSs usually run on the same hardware configuration as a worker, which have only one or two network interfaces. This implies incast congestion from their high

Network	CC	CS	NCC	NCS
ResNet 269	122	31	140	17
Inception	44	11	50	6
GoogleNet	40	10	46	6
AlexNet	1232	308	1408	176

Table 2.1: Estimated bisection bandwidth (Gbps) lower bound on the PS side for hiding communication latency in a small cluster of 8 nodes with GTX 1080 Ti.

bandwidth usage when serving multiple workers, starving the compute units. We profiled the training of multiple DNNs of different model sizes and computation-to-communication ratios. Our setup used 8 workers and 8 CS PSs. We observed *it was nearly impossible to eliminate communication latency in cloud-based training due to limited network bandwidth.* We estimated the minimum bandwidth requirement to fully hide communication latency in the network as follows: given a model size of M, and T time for each iteration, with N workers participating, the network should at least be able to send and receive model updates within the computation time (assuming infinitely fast PSs and that sending/receiving could fully overlap). Figure 1.7 gives an analytical lower bound of *per host bandwidth*, and Table 2.1 shows the required bandwidth for various DNNs: DNNs demand more bandwidth than mainstream offers (typically 10 Gbps).

One trivial solution would be to simply use interfaces with higher bandwidth. However, even in the best case, a single network interface is not capable of saturating memory or PCIe bandwidth. A single network interface also causes serialization delay and further imbalance across NUMA domains in a typical server.

2.2 Parameter Box Architecture

This section describes Parameter Box, our *balanced parameter exchange system.* We maintain that a centralized system, when architected properly, can provide high throughput, low latency, sufficient scalability for a rack, and low cost. We focus on the hardware side of Parameter Box, and in the next section, we detail the software side.

We prototyped Parameter Box using an off-the-shelf server platform that was configured to our requirements. Our goal was to balance IO and memory bandwidth; our prototype system had memory bandwidth of 120 GB/s and theoretical overall bidirectional IO bandwidth of 140 GB/s. To fully utilize resources, Parameter Box needed a matching network capability, which we provided by using multiple network interfaces. Figure 2.1 shows the resulting Parameter Box design. The system includes 10 network interfaces, each of 56 Gbps link speed, connected to a switch. This uses all PCIe bandwidth on our dual socket prototype

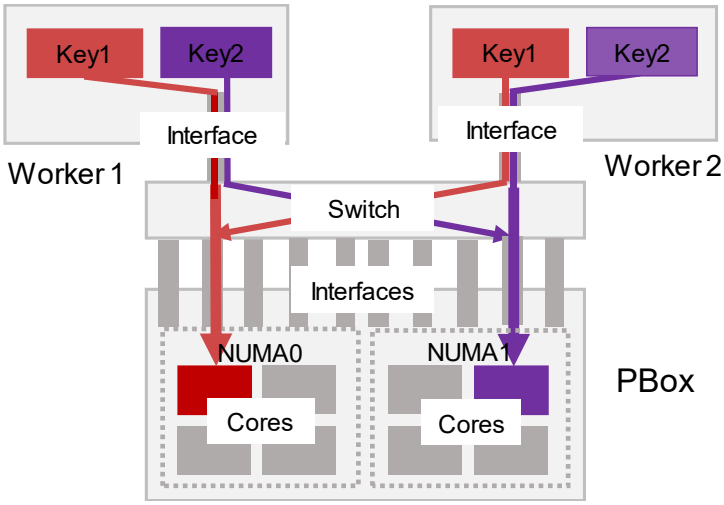

Figure 2.1: The Parameter Box architecture

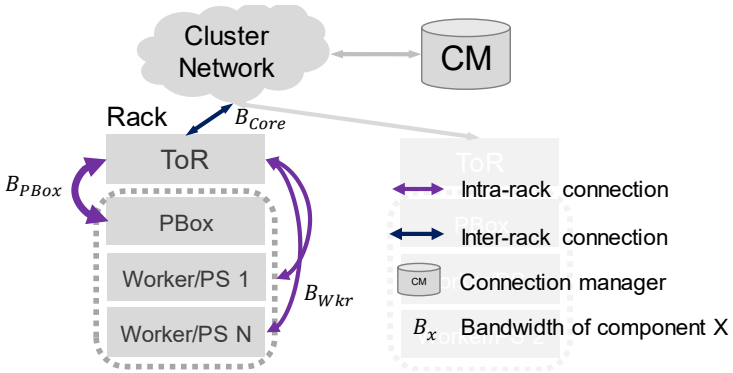

Figure 2.2: Parameter Box deployment scheme

and provides roughly 136 GB/s bandwidth once IB and PCIe framing overheads are taken into account, balancing IO and memory bandwidth.

2.3 Multi-Rack Deployment and Topology-Aware Reduction

When needed, Parameter Box can be used to upscale distributed training to datacenter levels. To extend service coverage of a single Parameter Box device, we associate a Parameter Box with a ToR during deployment, for two reasons. First, full bisection bandwidth is achievable for machines in the same rack, making it ideal for a central reduction entity as Parameter

Box, while oversubscription occurs between the ToR and the cluster network. Second, as we show later, a single Parameter Box has enough scalability for a typical rack of worker machines.

When provisioned in each rack (Figure 2.2), Parameter Boxes can form an array of sharded PSs, or run a *hierarchical reduction* algorithm for a training task that spans multiple racks through the coordination of a connection manager. Hierarchical reduction works in three steps: first, each Parameter Box centrally aggregates gradient updates from workers in the same rack; then, the Parameter Box nodes start cross-rack aggregation and compute globally aggregated gradients; finally, each per-rack Parameter Box runs an optimizer on this gradient and broadcasts the new weights back to local workers.

Hierarchical reduction trades off more rounds of communication for lower cross-rack traffic ($1/N$ with N-worker racks). We can determine when hierarchical reduction is potentially beneficial with the simple model below:

$$\frac{N(R-1)}{RB_{Core}} > max(\frac{N}{B_{PBox}}, \frac{1}{B_{Wkr}}) + C$$

where B_{PBox}, B_{Core} and B_{Wkr} are the bandwidths of a Parameter Box, the network core, and a worker, and R is the number of racks. When the condition is true, this means the time to perform cross-rack transfer is larger than the added latency of a two-level reduction, which consists of a per-rack local aggregation that happens in parallel and an inter-rack communication (with cost C) done with either sharded PSs ($C = \frac{r-1}{rB_{bn}}$, where $B_{bn} = min(B_{PBox}, B_{Core})$) or a collectives operation (e.g., $C \approx \frac{r-1}{rB_{bn}}$ with racks forming a ring). C can be directly measured, and B_{Core} can be effectively probed by using [60, 61].

	Local	2 nodes	4 nodes	8 nodes
TensorFlow	152	213	410	634
Caffe2	195	266	343	513
TF+Poseidon[161]	209	229	364	<648
MxNet	190	187	375	**688**

Table 3.1: Throughput (samples/s) of training ResNet 50 on major DNN training frameworks with a 56 Gbps network.

Chapter 3

PARAMETER HUB: STREAMLINED SOFTWARE STACK FOR RACK-SCALE PSS

Hardware alone solves only part of the problem. Existing frameworks cannot efficiently use the full hardware capability of Parameter Box (for example, TensorFlow and MxNet support multiple interfaces only by spawning multiple PS processes). The result is, even with ample communication resources, existing PSs failed to hide communication latency and struggled to scale. Table 3.1 shows that all major DNN training frameworks do not scale

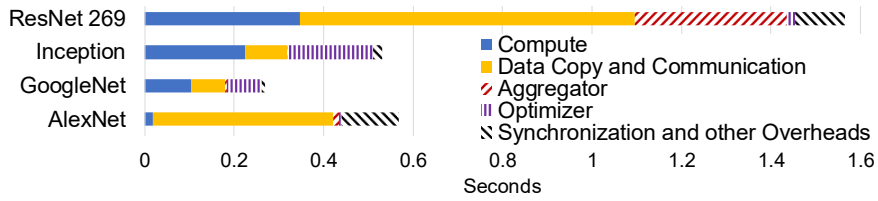

Figure 3.1: Progressive overhead breakdown of different stages during the distributed training pipeline for MxNet distributed training on a 56Gbps network. Link capacity accounts for a small fraction of the copy and communication overhead in this setting.

well with a 56 Gbps IPoIB network.

3.1 Inefficient Software Stack

We investigated the cause for MxNet by breaking down the overhead for each major component of a training iteration (legends of Figure 3.1). Since all stages overlap one another, and since ideally we would like early stages to fully hide the latency of later stages, we show *progressive overhead* in Figure 3.1: we gradually turned on different components in the MxNet distributed training pipeline, and each segment shows the *additional overhead that previous stages could not hide*. Specifically, the compute segment shows how long the GPU is active; the data copy segment shows the additional overhead of turning on distributed training without aggregation and optimization; the aggregation and optimization segments show additional overheads of enabling them in that order; and the "other" overheads segment includes synchronization and overheads that are not specific to a single component. We explain the overhead for some components:

Data copy: each layer's parameters were copied to and from OS buffers 4 times during parameter exchange.

Aggregation and optimization: MxNet's approach to achieving parallelism in these operations did not achieve high throughput in our measurements.

Synchronization: MxNet's dispatcher thread needs to synchronize access with ZMQ threads, aggregation threads and an optimization thread via shared queues, leading to bad locality and increased synchronization overhead.

3.2 Parameter Hub Software

Based on above findings, we propose Parameter Hub, an optimized PS implementation that reduces framework overhead with software optimizations. With Parameter Hub, we aim to:

1. Minimize gradient/model communication overhead.
2. Enable efficient gradient processing and overlap with communication.

We now software optimizations that benefit different stages in distributed training across all common PS configurations.

3.2.1 Network Stack Optimizations

We sought to mitigate data movement latency with zero-copy and kernel bypass. We chose InfiniBand (IB) since we were already familiar with the Verbs API, and it is available in major cloud providers [93]. Note that similar results could be achieved over Ethernet using RoCE, DPDK or other frameworks. We followed the guidelines from [71]; we tried two and one-sided RDMA, and two-sided send/receive operations and found similar performance in our workload. We briefly highlight some implementation details:

Minimal Copy: Leveraging InfiniBand's zero-copy capability, the only required data copy is between the GPU and main memory. When one GPU is used, this can be eliminated with GPU-Direct RDMA on supported devices.

NUMA-Aware, One-shot Memory Region Registration: Since a worker can operate on only one model update at a time, it is sufficient to allocate one read buffer (for the current model) and one write buffer (for update reception) for the model. To minimize InfiniBand cache misses, Parameter Hub preallocates all buffers in the NUMA domain where the card resides as a contiguous block.

Minimal Metadata: To maximize bandwidth utilization and minimize parsing overhead, Parameter Hub encodes metadata (such as callback ID and message opcode) into InfiniBand's queue pair number and immediate field. This saves Parameter Hub an additional PCIe round trip (from IB send scatter/gather) to gather metadata when sending messages.

3.2.2 Gradient Aggregation and Optimization

Gradient aggregation could occur in the CPU or GPU [32]. Here, we posit that the CPU is sufficient for this job. Aggregation is simply vector addition: we read two floats and write one back. With our typical modern dual socket server, if we keep our processors' AVX ALUs fed, we can perform 470 single-precision giga-adds per second, requiring 5.6 TB/s of load/store

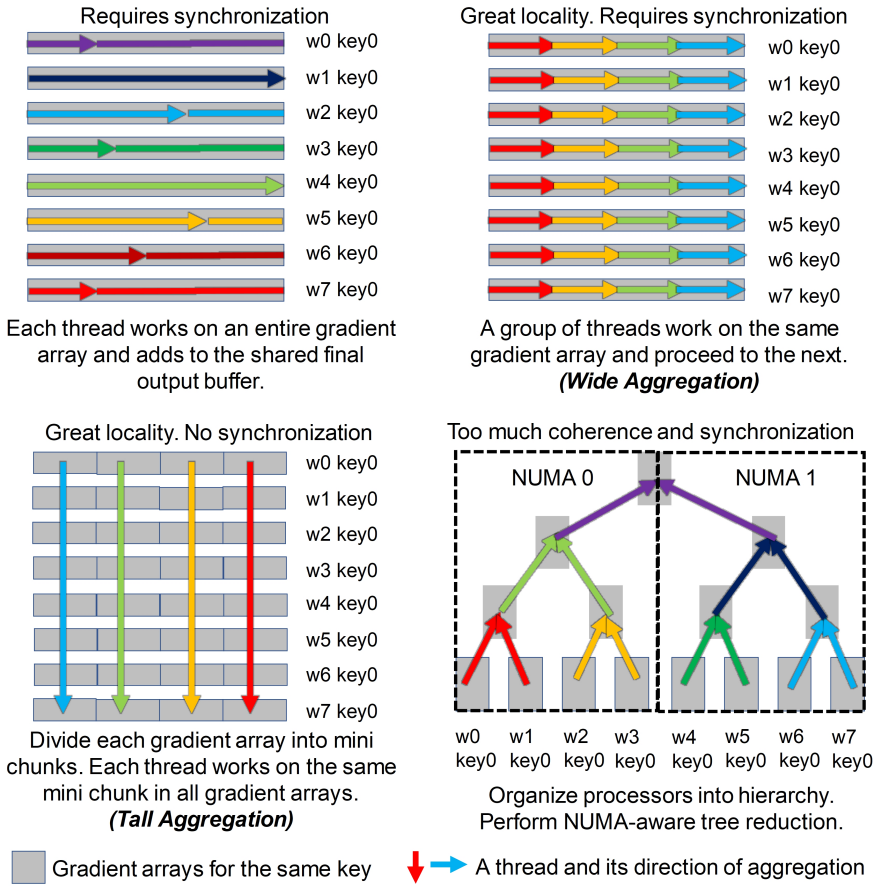

Figure 3.2: Ways of gradient aggregation. A thread (arrow) aggregates over the array (gray rectangle) of gradients from a worker.

bandwidth. But the processors can sustain only 120 GB/s of DRAM bandwidth, making aggregation inherently memory bound. Thus, copying gradients to a GPU for aggregation is not helpful.

There are many ways to organize threads to perform aggregation. Figure 3.2 shows four options we prototyped, assuming gradient arrays are available at once. We found that the best performance was achieved using the two discussed below; other schemes suffered from too much synchronization overhead, poor locality and/or high latency.

Wide aggregation is typical to systems like MxNet that call BLAS routines for linear

algebra. In these systems, a group of aggregation threads process one gradient array at a time; each thread works on a partition of that array.

A variation of wide aggregation is *tall aggregation*, which chunks a gradient array into mini-chunks of predefined sizes; each thread works independently to process the same chunk across all gradient arrays for a given key. This is the preferable way to organize threads for many reasons. First, gradient arrays do not arrive instantly. For a large key (e.g., a fully connected layer), aggregation and optimization cannot start for wide aggregation until the key is fully received; for tall aggregation, the process can start as soon as the first chunk is received. Second, in wide aggregation, it is challenging to balance the number of threads dedicated to aggregation and to optimization, let alone partitioning threads to work on different keys since they can arrive at the same time; thread assignment for tall aggregation is natural. Third, wide aggregation induces queuing delays: it effectively processes one key at a time versus tall aggregation's many "mini-queues." Fourth, wide aggregation puts many threads to work in lock-step on pieces of data, which incurs non-trivial synchronization overhead; tall aggregation requires no coordination of threads as aggregation is an element-wise operation.

Parameter Hub tracks the number of currently aggregated mini-chunks for a given key. When a chunk is received from all workers, it can be optimized. This step is natural in Parameter Hub: the thread that aggregates a particular chunk also optimizes that chunk. As a result, Parameter Hub's aggregation and optimization scheme effectively maps a particular chunk to a single core (since Parameter Hub pins threads to cores). On the other hand, MxNet uses wide optimization: when a key is fully aggregated, another set of threads is launched to perform aggregation. No overlap occurs between key aggregation and optimization.

We explored the benefits of caching by implementing two variants of each aggregator and optimizer: one using normal cached loads and stores, and one with non-temporal prefetches and stores. We found it beneficial to cache both the model and gradients. Parameter Hub's aggregators and optimizers are fully extensible: implementations that comply with

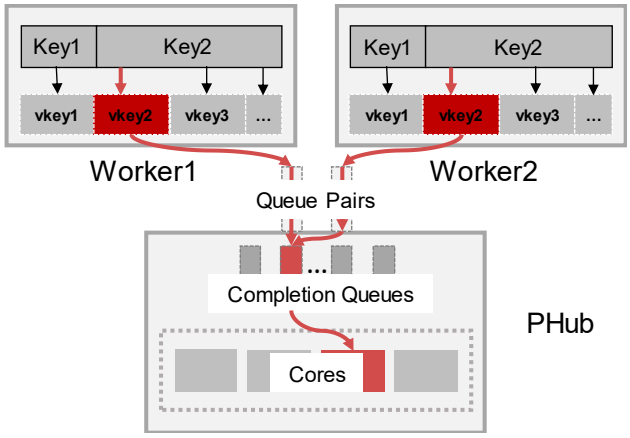

Figure 3.3: The process of mapping a chunk to a core in Parameter Hub using fine grained key chunking. Keys are chunked into virtual keys. The highlighted key is delivered to a highlighted (fixed) core through a highlighted (fixed) queue pair and completion queue.

Parameter Hub's API can be used during runtime.

3.2.3 Fine-grained Key Chunking

Chunking in Parameter Hub differs from other systems in key ways. Initially, our goal is to balance load at a fine-grained level across cores and interfaces rather than across server shards: chunking is turned on even when a centralized PS is used. Next, we would expect our optimal chunk size to be the smallest message size that can saturate network bandwidth, whereas systems like MxNet prefer larger key chunk sizes to avoid excessive thread synchronization overhead. In fact, Parameter Hub's default is 32KB, while MxNet's is 4MB. Finally, key chunking enables another important optimization: the overlapping of gradient transmission with aggregation and optimization. Aggregation starts only after a key's entire gradient array is received; and for large layers, this adds significant delay. With small key chunks, Parameter Hub enables "streaming" aggregation and optimization.

3.2.4 Mapping a Chunk to a Core

Parameter Hub's assignment of chunks to cores is computed during initialization. At that time, the set of all keys is sharded across the cores and interfaces available on PS nodes. A specific chunk is always directed to a particular queue pair, which is associated with a shared completion queue on the chunk's core. All message transmission, reception, and processing for that chunk is done on that core. Cores do not synchronize with each other. Once processed, a chunk is transmitted back to the workers on its originating path. The worker side of Parameter Hub assembles and disassembles a key, a process that is transparent to the framework.

Parameter Hub's chunk assignment scheme provides significant locality benefits. The same key likely arrives around the same time from multiple workers; the associated aggregation buffer is reused during this period. The scheme also encourages locality in the InfiniBand interface in the queue pair and memory registration caches.

This scheme imposes challenges in balancing load across cores, queue pairs and completion queues. Parameter Hub uses a 4/3 approximation set partition algorithm to balance each component's workload at each level, which produces practically balanced assignments in our experiments. Parameter Hub's chunk mapping mechanism is summarized in Figure 3.3.

3.3 The Parameter Hub Service API and Interoperability with other Frameworks

Parameter Hub's API is designed for compatibility with multiple DNN training frameworks. Workers use Parameter Hub by first calling `PHub::CreateService` on the connection manager. This sets up access control and a namespace for the training job and returns a handle. The client side uses the handle to finish setup. Parameter Hub uses the namespace and an associated nonce for isolation and access control.

Jobs call `PHub::ConnectService` to rendezvous servers and workers, exchanging addresses for communication. This call replaces `Van::Connect` in MxNet, `Context::connectFullMesh` in Caffe2 and `GrpcServer::Init` in TensorFlow. `PHub::InitService` causes the current Pa-

rameter Hub instance to allocate and register receive and merge buffers. Parameter Hub also authenticates each worker's identity using the nonce. Authentication is a one-time overhead and once a connection is established, Parameter Hub assumes the remote identity associated with that address/port/queue number does not change during training.

Parameter Hub's functional APIs include standard synchronous or asynchronous `PHub::Push/Pull` operations that are used in TensorFlow (`GraphMgr::SendInputs/RecvOutputs`) and MxNet (`KVStoreDist::PushImpl/PullImpl`). Parameter Hub also includes a fused `PHub::PushPull` operation that perform a push, waits until all pushes are complete, and pulls the latest model. The fused operation often saves a network round-trip as push and pulls are frequently issued consecutively. This operator can serve as a drop-in replacement for Caffe2's `Algorithm::Run`.

3.4 Interaction of Parameter Box and Parameter Hub

Parameter Hub takes full advantage of Parameter Box by extending the chunk-to-core mapping scheme, ensuring balance across interfaces and NUMA domains. Parameter Hub further guarantees no inter-processor traffic on Parameter Box, and completion queues and queue pairs in an interface card are used by only one core in the same NUMA domain to promote locality and avoid coherence traffic. In essence, Parameter Box forms micro-shards inside a box.

Chapter 4

PARAMETER LINK: DISCOVERING AND EXPLOITING DATACENTER NETWORK LOCALITY FOR EFFICIENT CLOUD-BASED DISTRIBUTED TRAINING

So far we have focused on accelerating distributed learning at a rack scale, by customizing hardware configuration, software stack, and network interconnect. While highly effective, not all of the optimizations are universally applicable, for example, in a public cloud environment. Further, even if all optimizations are applied, they do little to address cloud specific bottlenecks.

4.1 Specific Challenges in Public Clouds

Two major challenges exist in cloud-based training.

Non-uniform link bandwidth. Host-to-host bandwidth in the cloud is non-uniform due to the hierarchical structure of the datacenter (§1.1.3). Figure 4.1 shows a pairwise bandwidth probe of 32 VM nodes in EC2 and Azure, in the same availability zone/datacenter. In both cases, faster pairs can deliver more than 2x the throughput of slower pairs.

Volatile traffic. The performance variability in the public cloud is well known [45, 64, 88]. Although mechanisms for performance isolation have been proposed [132, 133], we still observe interference from other workloads, leading to volatile latency and aggregation performance (Figure 4.2).

4.1.1 Inefficiencies in Existing Approaches

We motivate our design by analyzing why some existing approaches do not perform optimally, as they rely on assumptions that aren't typically valid in the datacenter setting.

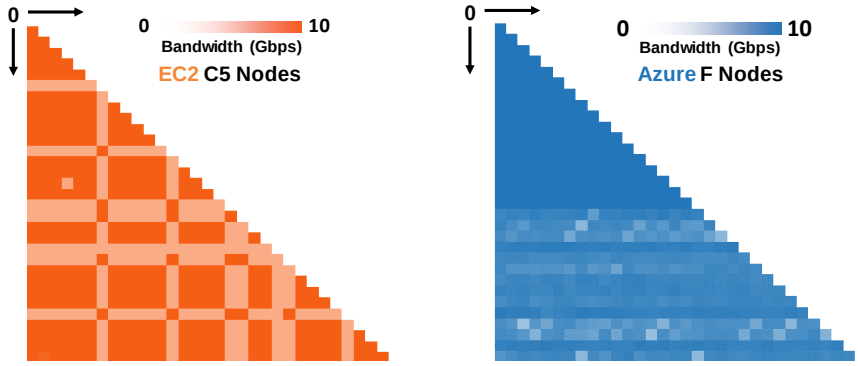

Figure 4.1: Pairwise bandwidth probes with 32 EC2 C5 and Azure F instances show non-uniform link bandwidth.

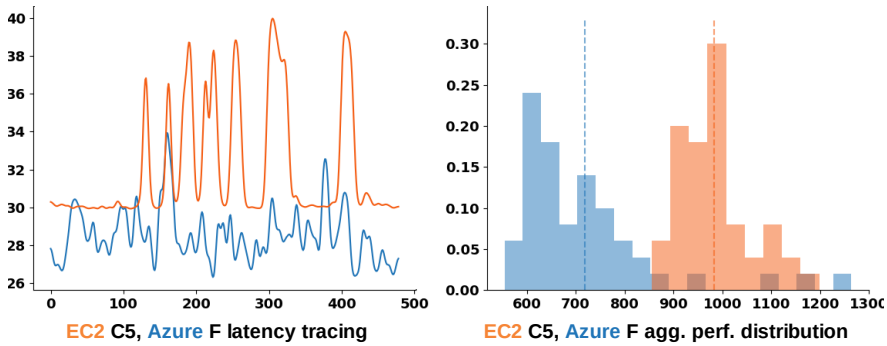

Figure 4.2: Left: 8 hour latency (us) tracing (1 minute average) between two VMs on both clouds show steep latency fluctuations due to volatile cloud traffic. Right: Wide performance distribution of the same, periodically launched Gloo aggregation task on both clouds.

Figure 4.3 shows a theoretical analysis of widely used communication patterns. PS (a) and popular choices of CA such as halving-doubling (b) and ring (c) are shown in a setup where nodes (0-3, enclosed in a circle) are spread equally among two clusters (purple and gold). The left side of the figure shows patterns that achieve optimal locality in the setting by exchanging data among nodes with high locality (high-performance links in green) while minimizing transfers over the bottleneck links (slow links in red). The right side shows alternative reduction routes with poor locality. All patterns achieve the same result, but with different efficiency.

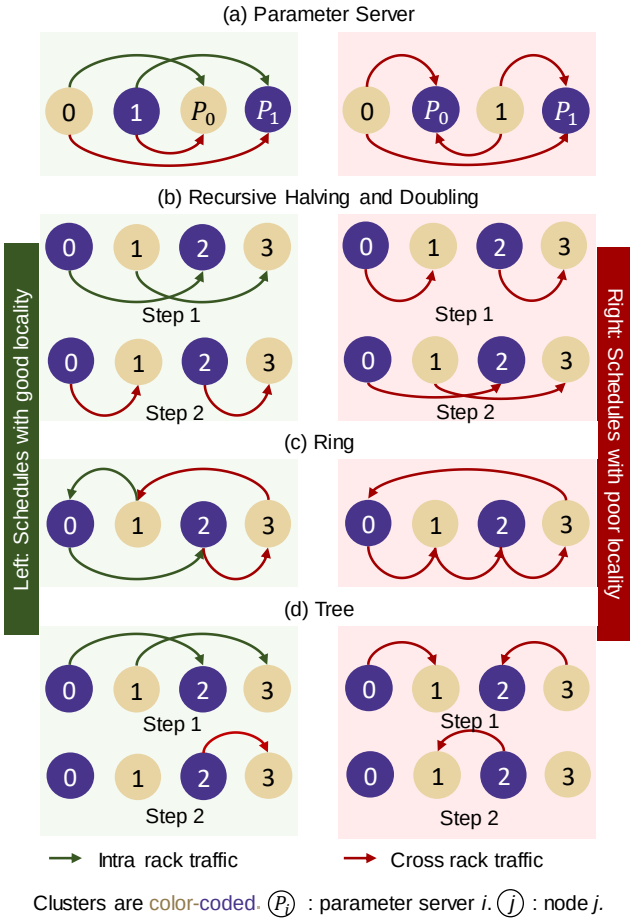

Figure 4.3: Existing aggregation approaches can suffer from poor locality if not taking physical network topology into account.

The problem with poor locality happens when the communication pattern in the algorithm is not optimally aligned with physical topology. Mapping logical ranks to physical hosts in a locality-preserving way is contingent on awareness of the physical network structure. Hence, *topology-awareness* is crucial for efficient aggregation in a datacenter network.

Even with careful mapping, not all algorithms work optimally in the datacenter environment. Table 4.1 summarizes network characteristics of these algorithms, with a simplified, flattened datacenter network topology model where nodes are simply placed in different racks. Centralized PSs are known to suffer from *incast* congestion and do not scale to a

large number of workers [62, 49]. Sharded PSs incur high cross rack traffic. CAs usually trade off lower per-link traffic on the wire with more rounds of communication, which is not suitable when the latency is high. Tree reduction inherits the problems of both PS and CA: high fan-out causes incast problems; a low fan-out adds more rounds. Ultimately, we need an algorithm that bounds communication steps, takes advantage of fast links, and localizes traffic to avoid interference from competing traffic.

Name	Rounds/Hops	Bytes on Wire	XR Bytes
PS (fully sharded)	2	$2(NC-1)S$	$2N(C-1)S$
Halving doubling	$2log_2NC$	$2NCS$	$2C\frac{N-1}{N}S$
Ring-Chunked	$2NC-1$	$(2NC-1)S$	$(2C-1)S$
Tree (fan out = C)	$2(log_CN+1)$	$\approx 2C\frac{CN-1}{C-1}S$	$\approx 2C\frac{(N-1)}{C-1}S$
2-level hierarchical	4	$2S(NC-1)$	$2(C-1)S$

Table 4.1: Network characteristics of various algorithms featuring rounds of communication (Rounds), minimum total traffic (Bytes on Wire), and minimum cross rack traffic (Min XR Bytes, corresponding to red arrows in Figure 4.3) to allreduce with NC nodes on C racks, each with N nodes. Each node has a buffer of size S bytes. PS and aggregators in HA are colocated and sharded.

4.1.2 2-level Hierarchical Aggregation (2LHA)

HA is not new, but most applications of 2LHA are in contexts where the network topology is known. HA does not reduce the total amount of data transferred on the wire, but it can create more *localized* traffic and avoid slow links.

One important parameter in HA is the number of levels. Similar to [3], we used a 2-level HA based on our domain knowledge of datacenter networks, that oversubscription mostly hurts at the rack level. Thus, by separating inter- and intra-rack aggregation, we can best capture the static aspect of locality and minimize latency. The use of more levels (> 2) suffers from higher latency and volatile performance, as messages need to traverse multiple

links with unpredictable latency, but provides no benefit compared to PS if links don't have enough non-uniformity (e.g., in the same cluster).

2LHA partitions nodes into different groups (clusters) based on their affinity. 2LHA starts by chunking the buffer across members in the same group. For each chunk, a node is designated as the local master (LM) for that group for aggregating locally. One of the LMs across all groups is chosen as the global master (GM) for global aggregation. Visually, the reduction trees of all chunks form a 2-level forest. Communication for 2LHA is done in the following steps:

1. Each group member sends all chunks to their respective LMs (intra-group traffic only).
2. LMs in all groups send per-group aggregated chunk to the GM for global aggregation (inter-group traffic only).
3. The GM aggregates the chunk, then uses the reversed routes for propagating the globally-aggregated chunk back to the LMs.
4. The LMs fan out the globally aggregated chunk to all group members.

2LHA is described here as a two-phase process for simplicity, but the intra- and inter-group aggregation can overlap. Effective 2LHA also requires load-balanced LM and GM assignments within and across groups. Later we provide an implementation that satisfies these. Table 4.1 shows the desirable properties of 2LHA. Compared to CAs, the number of rounds in 2LHA does not increase with the number of nodes and, compared to PSs, it requires significantly less cross-rack bandwidth.

4.2 Design and Implementation of Parameter Link

We now describe Parameter Link, an optimized, topology-aware, and dynamic system that leverages HA for efficient cloud-based training. To optimally utilize datacenter networks, Parameter Link must address the major challenges highlighted previously. Parameter Link uses three components to achieve this.

- ProbeEmbed: a network probing and clustering approach to capture physical locality in the datacenter network. ProbeEmbed groups nodes based on their physical affinity, so

intra-group links have better communication performance than inter-group links.

- AggEngine: a high-performance implementation of 2LHA that is codesigned to take advantage of deep learning properties. AggEngine uses clustering information to distribute the aggregation workload efficiently and execute the aggregation schedule.
- Autotune: a mechanism that tracks training performance and adjusts the current GM and LM assignments to adapt to changes in the network conditions.

4.2.1 Capturing Network Locality with ProbeEmbed

For accurate network topology discovery, ProbeEmbed must probe quickly and should not rely on knowledge of a particular datacenter. ProbeEmbed: (1) probes communication links between nodes to measure pairwise node distances, (2) denoises probed distances, and (3) clusters nodes.

Running ProbeEmbed probes ProbeEmbed starts by issuing measurements to identify communication locality and determine pairwise node distances. Distance is defined using universal networking concepts, like latency or inverse bandwidth. ProbeEmbed uses two different probes: an inhouse DPDK-based [40] probe to provide near bare-metal latency measurements for supported VMs on Azure and EC2, and iPerf [65]. ProbeEmbed runs these networking probes one-to-one. $O(N^2)$ time would be required to probe N nodes if run sequentially. To accelerate this process, ProbeEmbed picks as many pairs (up to $\frac{N}{2}$) as possible in each round without having a node appear twice, to avoid interference from concurrent tests. This allows ProbeEmbed to probe in $O(N)$ rounds.

ProbeEmbed derives pairwise distances with probe results (in case of bandwidth measurements, bandwidth are converted to distance by taking the inverse [126]). ProbeEmbed then proceeds to *denoise* the collected data.

Denoising probe data with embedding ProbeEmbed embeds nodes in a Euclidean coordinate space, obtaining a set of coordinates whose distances agree with the probed

distances. This works to:

1. Denoise measurements by leveraging Euclidean space to approximate the physical location of nodes.
2. Obtain a set of "virtual coordinates" for a clustering algorithm to identify groups.

To embed nodes, we identify node coordinates ($\mathbf{v_i}$ and optional h_i) that minimize the following objective:

$$\sum_{i=1}^{n}\sum_{j=1}^{i-1}\left((d_{i,j})^{\alpha}+\mathbb{1}_h\left[h_i+h_j\right]-p_{i,j}\right)^2 \tag{4.1}$$

where n is the number of nodes, $d_{i,j}=\left\|\mathbf{v_i}-\mathbf{v_j}\right\|_2$ is the Euclidean distance between embedded node coordinates for nodes i and j, $p_{i,j}$ is their probed distance, parameter α takes a value between 1 and 2, h_i is a non-negative startup cost parameter for node i, and $\mathbb{1}_h$ is a switch for h_i. We use the Adam algorithm [75] to optimize this.

ProbeEmbed embeds VM nodes in a coordinate space that preserves the probed distances between VM nodes. The denoising effect of the embedding process stems from its tendency to keep mutually close nodes together, which enforces our domain knowledge that VM nodes that are close to one particular reference VM node are probably close to each other as well in the datacenter. Thus, this effect has a correcting influence when the mutual-closeness property is violated by a particular observation but is observed in a majority of nodes. A lower number of embedding dimensions strengthens this effect.

α tunes how longer distances are treated in the Euclidean space: $\alpha=1$ fits the embedded distances to probe distances exactly. For $\alpha>1$, we can achieve increased compaction of distance while maintaining relative distance order. This effect is desired because small physical distances can be magnified disproportionately in the probe due to competing traffic. Setting $\alpha=2$ causes the long probed values to be "compacted" more than the smaller ones (but it never changes the relative order of distances). Figure 4.4 suggests empirically, a larger α pushes VM nodes with short distances even closer on the embedded plane, leading to more consistent clusters with higher adjusted mutual information [145] (0.59 vs 0.76) across 100 runs.

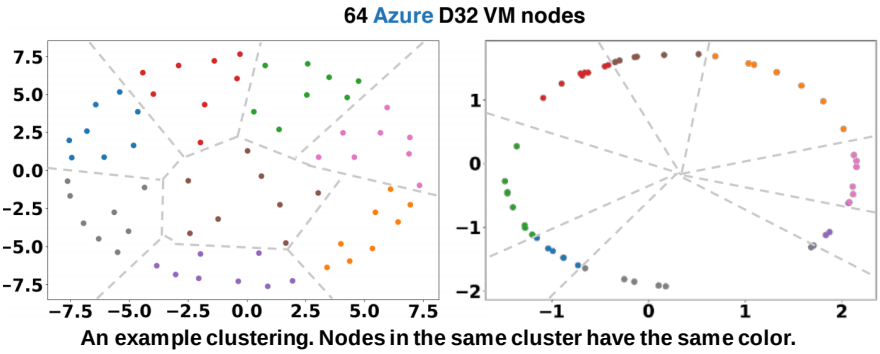

Figure 4.4: Effect of α: $\alpha = 2$ (right) generates more consistent clusters (higher AMI across 100 generated clusters) compared to $\alpha = 1$ (left)

Inspired by [34], ProbeEmbed includes an optional parameter h_i, the node-specific, network-agnostic, fixed latency of sending a packet (e.g. traversing the operating system network stack).

Multiple clusters are generated at once, and ProbeEmbed favors clusters with smaller diversity in the sizes of groups. If there is a tie, ProbeEmbed makes a random choice.

Grouping Nodes for 2LHA We now outline how ProbeEmbed partitions VM nodes into groups for 2LHA. The goal is to generate groups that are (1) balanced, so no single group becomes a bottleneck and (2) cohesive, so VMs in the same group have good locality. We first compute the number of groups to generate, then determine the members of each group.

GMs are more likely to be the bottleneck during 2LHA, as they must receive and aggregate messages at both levels. For a uniform key distribution, the following term approximates the bytes-on-wire sent or received by a GM:

$$b = O\left(\frac{n}{k} + k\right) \tag{4.2}$$

with n total VMs, and k groups. The GM sends and receives a message from each node within its group ($\frac{n}{k} - 1$) for local aggregation, and from every other group ($k - 1$) for global aggregation. This expression achieves a minimum at $k = \sqrt{n}$, giving a natural choice for group count.

Once group count is selected, we use a constrained k-means clustering algorithm with k-means++ initialization [22, 15] to generate balanced, locality-preserving groups. This accepts a minimum cluster size and the number of groups to generate as input. For perfect balance, both parameters are set to $\sqrt{n}$.

Enforcing perfect balance is not optimal in cases where VMs are naturally clustered in almost balanced but distant clusters, because in those cases some group can contain a distant member which could be assigned to a much more cohesive group with a slight imbalance, forcing an onerous bottleneck on the local aggregation step. Thus, we include a parameter, *balance elasticity*, which enables a slight imbalance among clusters. We empirically found best results with values between 1.0 (perfectly balanced) and 2.0 (each group has at least $\sqrt{N}/2$ nodes).

In order to evaluate the performance of ProbeEmbed, we also define Balanced Random, a grouping method for 2LHA that operates without considering probe distances and simply produces $\sqrt{n}$ groups of size $\sqrt{n}$ uniformly at random. This is used later as a baseline.

4.2.2 Efficient HA with AggEngine

AggEngine transforms grouping information from ProbeEmbed into a hierarchical reduction plan and efficiently executes it. AggEngine takes the crux of Parameter Hub and applies them to a TCP context to accommodate for the cloud environment where InfiniBand is not available. AggEngine supports various communication backends, including TCP, RoCE [138], iWarp [92], and InfiniBand (supported through Parameter Hub). To avoid repeating similar contents, we only highlight the difference of AggEngine and Parameter Hub.

Generating an Aggregation Plan AggEngine chunks buffers into 64KB segments for better load-balancing across processor cores and overlapping of the transmission of gradients with aggregation.

Since AggEngine is bottlenecked by the slowest inter-group transfer, which in turn is bottlenecked by the slowest intra-group transfer, AggEngine assigns chunk GMs and LMs

such that each group (or node) has a number of GMs (or LMs) proportional to its cardinality. AggEngine uses an approximation set partition algorithm to achieve this.

AggEngine then generates a schedule that executes the steps in §4.1.2 for each chunk. A schedule consists of a set of chunk-action pairs, where action is one of the following[1]:

- **SendTo(nids)**: send the content in the current *merge buffer* to the list of nodes specified in *nids*. SendTo is a non-blocking operation, and its status is inferred by whether subsequently anticipated data is received.
- **ReceiveFrom(nids)**: block until the chunks from *nids* are received and aggregated into the merge buffer.
- **Fetch**: notifies AggEngine that a framework-supplied buffer is ready to be processed.
- **Deliver**: writes the content in the merge buffer back to the framework-supplied buffer.

A schedule is represented as a DAG where dependencies are edges and nodes are primitives, avoiding false dependencies between local and global aggregation.

Executing an Aggregation Schedule AggEngine first performs rendezvous with an out-of-band mechanism (e.g., Redis), establishing multiple connections per pair of VM as cloud providers can restrict per-stream bandwidth [10]. AggEngine preserves intra-node locality by maintaining a load-balanced map from a chunk to a connection, and then by further associating the connection to a particular processor core [115].

We now focus on how AggEngine efficiently supports the four actions, hiding communication latency and avoiding excessive synchronization.

When a framework calls `reduce(chunk)`, AggEngine retrieves the thread ID, suspends it, and enqueues `chunk` to the ready queue. Its worker threads poll the ready queue to retrieve `chunk` and transition the buffer into the Fetch state, copying gradients from the supplied address, then set the state of buffer to SendTo.

SendTo is an asynchronous operation that simply enqueues data to a send FIFO queue. A cursor is used for each TCP connection if a send operation cannot finish. AggEngine enforces

[1]With these action primitives, AggEngine can support arbitrary reduction graphs.

Property	AggEngine Optimization
Fixed comm. pattern	No explicit acknowledgement
Fixed buffer size	Minimal metadata
One reduction per layer per iteration	Only 2 merge buffers per layer; Eagerly accepts chunks
Training is stochastic	Switching plans quickly and cheaply

Table 4.2: Codesigning AggEngine with training workload.

that the order of bytes on the wire corresponds exactly to the order in which SendTos are issued. This saves metadata overhead as only a 4B integer per flow that encodes the chunk ID is required.

SendTo is followed by ReceiveFrom. AggEngine allows streaming aggregation for each buffer in ReceiveFrom state to a *merge buffer*. A counter is incremented when a chunk is fully received from a peer, and when the counter reaches the target, this step concludes. AggEngine transitions current buffer state to SendTo or Deliver based on schedule.

The last step of a schedule is Deliver, where AggEngine copies the final value to the framework-supplied address. AggEngine then wakes up the thread that called `reduce`. AggEngine alternates between two merge buffers per chunk for synchronous training to overlap local computation on a chunk with transfers of that chunk to peer nodes.

Table 4.2 summarizes how AggEngine is designed to take advantage of properties in the distributed training workload to lower its overhead.

4.2.3 Reacting to Network Changes with Autotune

Autotune collects performance information from AggEngine and watches for sudden changes in link conditions, reflected by the current training speed. The goal of Autotune is to dynamically compensate for link changes by redistributing LMs and GMs to VMs, so the time to finish an iteration is similar at both local and global levels.

A perfect initial LM and GM assignment is hard, even if we have bandwidth probe

measurements. Consider an aggregation plan S, where the *effective bandwidth* of node i to j while running aggregation S is $BW(i,j)$. Clearly, finding the best S analytically relies on $BW(i,j)$ to be precisely measured or modeled, but $BW(i,j)$ has a circular dependency on S itself. Autotune thus makes approximations when optimizing the assignments.

At a high level, Autotune works in two phases: (1) a *Quicktune* phase where a one-shot, global adjustment of GM and LM assignments is done to adapt to the network change immediately; and (2) a *Finetune* phase where Autotune uses a performance model to find the current performance bottleneck in the system, and moves GMs and LMs away from it in a stepwise, increasingly aggressive manner.

Quicktune Quicktune aims to minimize the maximum transfer time of each node. Quicktune can be best summarized formally as follows: let $GM(i,c) \in \{0,1\}$ and $LM(i,c) \in \{0,1\}$ be the boolean variables to be solved, which indicate whether node i is the GM or LM of chunk c. Let $G(i)$ be the group of node i, $|G|$ the number of groups and $S(c)$ the size of c in bytes. Our goal is to:

minimize

$$max(t_i = \frac{\sum_c S(c)(LM(i,c)|G(i)| + GM(i,c)|G|)}{\sum_n BW(i,n)})$$

subject to

$$\forall_{i,c} \quad GM(i,c) = 1 \implies LM(i,c) = 1$$

$$\forall_c \quad \sum_i GM(i,c) = 1$$

$$\forall_{c,g \in G} \quad \sum_{i \in G(g)} LM(i,c) = 1$$

Quicktune solves this with an approximation: it first distributes GMs to different groups, with the number of GMs assigned to each group proportional to the group aggregate bandwidth $\sum_{m \in G(i)} \sum_{n \notin G(g)} BW(m,n)$, then distributes LMs inside each group to different members in a similar fashion, using aggregate per node bandwidth.

Finetune Quicktune is limited as it assumes constant effective bandwidth across different schedules and ignores node balance. Finetune, however, amends this by gradually *evolving the current schedule*, using both the currently measured $D(i,j)$ and $B(i,j)$ to pinpoint the current bottleneck node in the system, and then moves away its load while maintaining balance based on *blame*. Blame for node i ($B(i)$) has two major weighted parts: *time* $t(i)$ and *imbalance* $l(i)$. Autotune collects per connection stats including link $RTT(i,j)$, bandwidth $BW(i,j)$ through the OS [150, 89], and computes $t(i) = max_j RTT(i,j) + \frac{\sum_j D(i,j)}{\sum_j BW(i,j)}$ and normalized $\hat{t(i)} = \frac{t(i)}{mean_n t(n)}$. Further, we let $l(i) = \sum_j D(i,j)$ and weighted $\hat{l(i)} = \frac{l(i) - min_n l(n)}{max_n l(n) - min_n l(n)}$. Finally, blame is defined as $\beta\hat{l(i)} + \gamma\hat{t(i)}$.

With blame for each node available, Finetune attempts a move of a single GM (or if not available, an LM) from the node with highest blame to the lowest, if $\frac{max_i B(i)}{min_i B(i)} > \epsilon$ (a configuration parameter). If Finetune repeatedly identifies the same bottleneck node, it moves exponentially more LMs and GMs in each step.

Autotune uses a central daemon to collect performance metrics and generate new schedules, and is triggered by a sudden change (e.g., larger than 20%) in training performance. Autotune signals AggEngine to install the new schedule.

Chapter 5

PLINK COLLECTIVES: TOWARDS CLOUD-AWARE COLLECTIVES WITH RANK REORDERING

We now shift our focus from building efficient parameter servers to another popular paradigm: collectives, used in popular training frameworks such as Caffe2 and Pytorch, where there is no longer role of servers and workers. We also address the need for specialization on alternative interconnects other than a fat tree topology in the datacenter, e.g., with a torus ring in Google TPU pods, because in these highly specialized environments, our optimizations highlighted earlier may not be sufficient.

5.1 Unoptimal use of Collectives Today

Unfortunately, achieving good collectives performance in a cloud environment is fundamentally more challenging than in an HPC world, because the user has no control over node placement, topology and has to share the infrastructure with other tenants. These constraints have a strong implication on the performance of collectives. As a result, the bottleneck of running these workloads with collectives on the cloud has shifted from computation to communication [159].

Consider a common practice of applying *allreduce* ring collectives, a popular algorithm, in the cloud context, where a randomly-ordered IP list (obtained through the provider) of VMs is used to form a virtual ring on which data is passed along, with i-th VM sending data to $i+1$-th VM. But do different ways of forming ring (through permutation of VMs in the list) exhibit the same performance? The answer is most likely no, as the ring that corresponds to shorter total hop cost will likely perform better (Figure 5.1). On the other hand, not all ways of forming rings achieve the same cost, because the *point to point communica-*

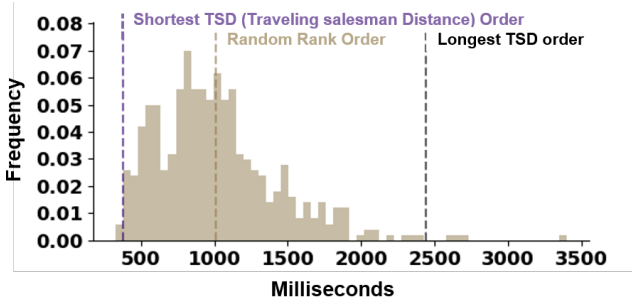

Figure 5.1: Performance distribution of *allreduce* task of 100MB data with ring algorithm varies widely with 500 random rank orders on Azure.

tion cost (bandwidth, latency, or collectively referred to as locality in this work) is different across VMs (Figure 4.1), due to the hierarchical structure of the datacenter network, and the dynamic nature of traffic from other tenants. Consequently, running collectives with a randomly-ordered list of VMs results in unpredictable and subpar performance.

Our work focuses on discovering a permutation of the IP list that exploits the network locality for efficient communication, in a completely transparent way, by minimizing the cost model of a given collectives parameterized with the actual hop cost. To do so, we need to (1) efficiently identify the underlying network bandwidth/latency constraints (or collectively, locality); (2) accurately build cost model for the collectives at hand; (3) effectively approximate the minimum of these complex cost functions.

This paper proposes ☁Collectives, a tool that uses accurate network probes to discover locality within the underlying datacenter network, and uses it to solve a communication cost minimization problem with constraints, with the rank of each VM as the unknowns. We use reordered ranks as input to unmodified communication backends in microbenchmarks including OMB [23], Nvidia NCCL, Facebook Gloo and real-world workloads of training deep neural networks with Pytorch/Caffe2 and gradient boosted decision trees using LightGBM [91, 72] and found a speedup of up to 3.7x in various *allreduce* operations and 1.3x in end-to-end performance across EC2 and Azure.

5.2 Design and Implementation

We now describe ☁Collectives, a tool that takes in a list of VM nodes and a target algorithm, accurately and efficiently probes their pairwise distance, and uses that information to construct a rank order of VMs that attempts to minimize the total cost of communication.

5.2.1 Cost Models for Collective Algorithms

☁Collectives builds a cost model $\mathbb{C}_{\mathbb{O}}$ for each popular algorithm used in collectives $\mathbb{O}$, parameterized with the number of participating nodes N and size S. This section details the cost models for popular algorithms. We use $c_{i,j}(S)$ to refer to the cost for transferring S amount of data from node i to j. We further define $MAX_{i=0}^{j}(f(i)) = MAX(f(0), ..., f(j))$. We assume N a power of 2 to simplify explanation, and allow arbitrary rank r to alias to canonical rank $(r + N)$ mod N.

Ring. The cost model of the ring algorithm is the sum of the cost of each hop when traversing the ring:

$$\mathbb{C}_r(N, c, S) = \sum_{i=0}^{N-1} c_{i,i-1}(S)$$

Having Doubling. The cost of halving doubling is the sum of costs for each round of communication, which in turn is the max cost of all communications in that round.

$$\mathbb{C}_{hd}(N, c, S) = \sum_{i=0}^{log_2 N-1} MAX_{j=0}^{\frac{N}{2}-1} c_{j,j+2^i}(\frac{S}{2^{i+1}})$$

Tree. The cost of running tree algorithms depends on the number of trees and how trees are constructed. The total cost is the maximum cost of all trees, which is in turn determined by the maximum cost of each subtree. We provide a cost model for a popular variant of tree algorithm: double binary tree as used in [109].

$$\mathbb{C}_{dbt}(N, c, S) = T(0, N - 1, S)$$

where $T(i, j, S)$ is expressed recursively:

$$T(i,j,S) = \begin{cases} 0 \textbf{ if } i \geq j \\ MAX(c_{\frac{i+j}{2},\frac{3i+j}{2}-1}(\frac{S}{2}) + T(i, \frac{i+j}{2} - 1), \\ c_{\frac{i+j}{2},\frac{i+3j}{2}+1}(\frac{S}{2}) + T(\frac{i+j}{2} + 1, j)) \textbf{ otherwise} \end{cases}$$

Similarly a mirrored tree is built by decrementing each node's rank in the tree without changing the tree structure.

BCube. The cost of running the BCube algorithm is similar to halving doubling, except in each round, each node communicates with $B - 1$ peers, instead of 1.

$$\mathbb{C}_b(N, c, S, B) = \sum_{i=0}^{log_B N-1} MAX_{j=0}^{\frac{N}{B}-1} MAX_{k=1}^{B} c_{j,j+kB^i}(\frac{S}{B^{i+1}})$$

5.2.2 Probing for Pairwise Distance

We need to determine values for $c_{i,j}(S)$ with end-to-end measurements. We first consider commonly used, simple linear model: $c_{i,j} = LAT_{i,j} + \frac{S}{BW_{i,j,S}}$ where $LAT_{i,j}$ being the one-direction latency from i to j, and $BW_{i,j,S}$ the bandwidth achieved with data size of S. There are immediate challenges of deriving $BW_{i,j}$ correctly: first, $BW_{i,j,S}$ varies depending on the size of packet size S being transferred: e.g., on a 10Gbps link it is unlikely to saturate full bandwidth while sending small packets. Figure 5.2 (left) shows the how bandwidth varies with the size of the buffer in a point to point *iperf* (TCP, using DCTCP [5]) test on two 30Gbps D64 nodes on Azure. It is cumbersome to create such profile for each pair of VMs; second, even if a profile like this is constructed, it may still fall short when multiple streams are competing for bandwidth: the streams do not share the bandwidth equally, but rather, one stream can consistently outperform the other in a long time trace, as shown in Figure 5.2 (right) with 3 D64 nodes on Azure, when both of them can achieve similar throughput when run individually. It seems intractable to derive an accurate BW given S and a set of competing streams. Third, many algorithms operate in chunked mode, allowing

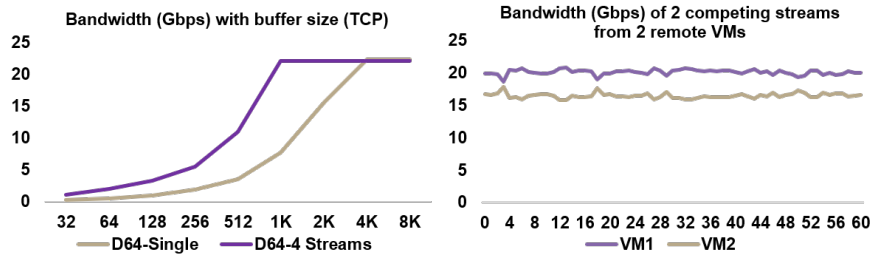

Figure 5.2: Left: TCP throughput depends on the buffer size and the number of concurrent streams. Right: while TCP is designed to be fair, an empirical 60s trace in the cloud shows the two streams connecting to the same VM from two other remote VMs do not share the bandwidth equally.

overlapping of sending (of processed elements) and receiving (of unprocessed elements), and the simple model does not capture this in the first place.

It is both beneficial and interesting to accurately model throughput behavior in a multi-stream environment [4, 100, 87, 55], but that subject is worthy of its own topic. Instead, we compromise by dropping the bandwidth component in the model, leaving only the latency component. The rationale behind this stems from the well-known theoretical TCP bandwidth model of $BW = O(\frac{MSS}{RTT\sqrt{p}})$ [90] given constant drop rate p and window MSS. The fact that higher latency induces lower bandwidth in TCP streams lets us approximate costs by only probing for latency.

To accurately and efficiently probe for pairwise latency, we built an in-house DPDK based echo tool, leveraging network enhancement provided by the clouds [94, 8]. Probing of N nodes can finish in N rounds, with each round probing for $N/2$ pairs of VMs. At round r, node i sends $10k$ probes to $(i + r + 1)$ mod N and responds to $10k$ probes from $(i - r - 1 + N)$ mod N sequentially. To derive an accurate reading, we take the RTT of 10th percentile. Each probe is a UDP packet with a 32-bit payload that encodes sequence number and round id for fault tolerance. When DPDK cannot be used, we use *fping*, a ICMP Echo-based latency probing tool. For each entry in c, we update $c_{i,j} \leftarrow MAX(c_{i,j}, c_{j,i})$ to make it symmetric.

5.2.3 Minimizing the Cost Model

We parameterize the cost model with values of probed c. To derive a rank ordering that minimizes $\mathbb{C}_{\mathbb{O}}$, we perform the following transformation: let set of variables $\mathbb{R}$ defined as $r_i, i \in [0, N-1]$ be a permutation of $[0, N-1]$ to be solved, and we replace each $c_{i,j}$ with c_{r_i,r_j}. We can then establish a bijection from the original rank ordering to the desired order $r_i \leftrightarrow i$ once r_is are solved. We flatten $c_{i,j} \leftrightarrow c'_{iN+j}$ to use theory of arrays to allow direct solving with conventional optimizing SMT solvers such as Z3 [36, 52].

Unfortunately, we find solvers inefficient, perhaps due to the non-convex, non-linear nature of the objective function and a large search space ($N!$). Instead, we take a two-stage process. The first step employs a range of stochastic search techniques such as simulated annealing [18], with a few standard heuristics (e.g., permuting a random sub-array, permuting random pairs) for obtaining neighboring states and a timeout. When the search returns with an initial result C_0, we generate an additional SMT constraint $\mathbb{C}_{\mathbb{O}} < C_0$ to better guide pruning for solvers. We let the solver continue to run for a few minutes, and we either find a better solution or will use C_0 as the final value. The end-product of this process is a rearranged list of VMs.

Chapter 6

EVALUATION

We now evaluate the effectiveness of Parameter Box, Parameter Hub, Parameter Link and ☁Collectives in their perspective environments in accelerating distributed training workloads.

6.1 Effectiveness of Parameter Hub and Parameter Box

We added support for Parameter Hub's API to MxNet, replacing its PS. We evaluated Parameter Hub by comparing it to MxNet's unmodified PS. We had five goals in our evaluation: (1) to assess the impact of Parameter Hub software and the Parameter Box hardware on training throughput, (2) to identify the importance of each optimization, (3) to determine the limits of Parameter Box, (4) to evaluate effectiveness of Parameter Box as a rack-scale service. and (5) to demonstrate the cost-effectiveness of the Parameter Hub.

6.1.1 Experimental Setup

We evaluated our system with 8 worker nodes and one specially configured Parameter Box node. The workers were dual socket Broadwell Xeon E5-2680 v4 systems and 64 GB of memory using 8 dual-rank DDR-2400 DIMMs. Each worker had a GTX 1080 Ti GPU and one Mellanox ConnectX-3 InfiniBand card with 56 Gbps bandwidth in the same NUMA domain. The Parameter Box machine was a dual socket Broadwell Xeon E5-2690 v4 system with 28 cores and 128 GBs of memory using 8 dual-rank DDR-2400 DIMMs. Parameter Box had 10 Mellanox ConnectX-3 InfiniBand cards, with 5 connected to each socket. Hyperthreading was disabled. Machines were connected with a Mellanox SX6025 56 Gbps 36-port switch.

The machines ran CentOS 7.3 with CUDA 8 and CuDNN 7 installed. Our modifications to MxNet and its PS (PS-Lite) were based on commit 2ce8b9a of the master branch in the

Name (Abbr)	Model Size	Time/batch	Batch
AlexNet (AN)	194MB	16ms	32
VGG 11 (V11)	505MB	121ms	32
VGG 19 (V19)	548MB	268ms	32
GoogleNet (GN)	38MB	100ms	32
Inception V3 (I3)	91MB	225ms	32
ResNet 18 (RN18)	45MB	54ms	32
ResNet 50 (RN50)	97MB	161ms	32
ResNet 269 (RN269)	390MB	350ms	16
ResNext 269 (RX269)	390MB	386ms	8

Table 6.1: Neural networks used in our evaluation. Time/batch refers to the forward and backward compute times for a batch.

PS-Lite repo. We built MxNet with GCC 4.8 and configured it to use OpenBLAS and enable SSE, the Distributed Key Value Store, the MxNet Profiler, and OpenMP. We used Jemalloc, as suggested by MxNet.

6.1.2 DNNs Used in the Evaluation

We evaluated Parameter Hub's performance by training state-of-the-art deep neural networks using reference code provided with MxNet. We implemented a cache-enabled optimizer using SGD with Nesterov's accelerated gradient method [104] and a cache-enabled aggregator for Parameter Hub. We chose a per GPU batch size of 32 when possible; for ResNet 269 and ResNext 269, we used 16 and 8, respectively, since 32 did not fit in the GPU. We did not use MXNet's GPU memory optimizations [26] because they slow down training.

Table 6.1 summarizes the neural networks used in our evaluation, which include both winners of the ImageNet challenge and other recent, popular networks. We used the reported model size from MxNet and measured the forward and backward passes time on a single GPU.

We report only training throughput in our evaluation since our modifications did not

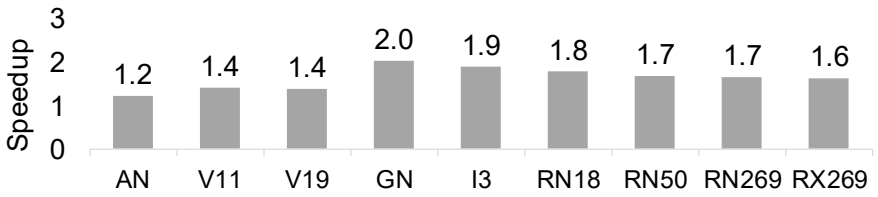

Figure 6.1: Speedup from a faster data plane that supports zero copy.

change accuracy: they did not change the computations that were performed. We trained multiple DNNs to convergence to verify this.

6.1.3 Training Performance Evaluation

We include multiple results to highlight the effects of different software and hardware optimizations on Parameter Hub's training performance. We measured training performance by comparing the total time of 200 iterations. We used two IB network configurations. This lets us compare training performance for two different compute/bandwidth ratios: (1) where GPUs were much faster than the network, and (2) with ample network bandwidth resources. In both setups, we used 8 workers.

Benefit of a Faster Data Plane

Figure 6.1 shows the performance of replacing the communication stack of the MxNet PS with a native InfiniBand implementation (MxNet) that had all optimizations enabled. This lets us see the benefit of switching to an optimized network stack without changing the PS architecture. We used our *enhanced baseline MxNet* in all the following evaluation.

Other Software and Hardware Optimizations

We now quantify further benefits from Parameter Hub's software and hardware optimizations. We used CS MxNet in this comparison. PShard results were obtained by running Parameter Hub software on each worker as CS PSs. Parameter Box results represent running Parameter

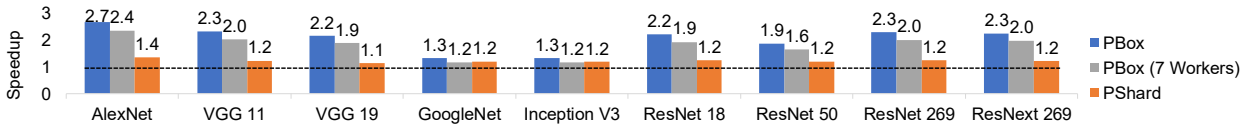

Figure 6.2: Training performance on a cloud-like 10 Gbps network. Results are normalized to sharded MxNet (*enhanced baseline*).

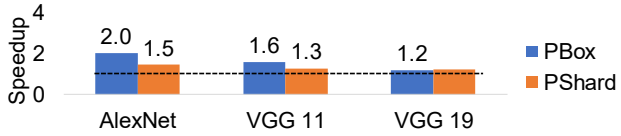

Figure 6.3: Training performance on a 56 Gbps network compared to MxNet (*enhanced baseline*). Computation speed bottlenecked training throughput for all but AlexNet and VGG.

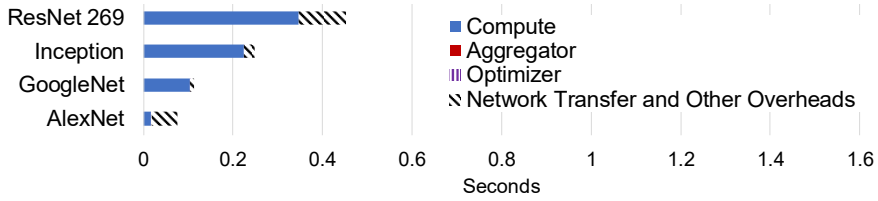

Figure 6.4: Progressive overhead breakdown of Parameter Hub. Compared to Figure 3.1, GPU compute time now dominates training time. Aggregator and optimizer have minimum overhead, and are barely visible.

Hub software on our single Parameter Box machine as a NCC PS. We omit results for NCS and CC PSs for clarity. They performed similarly to Parameter Box results.

Figure 6.2 shows training performance on a cloud-like 10 Gbps network, obtained by down-clocking our IB links. In this configuration, the ratio of GPU batch execution time to network serialization delays is such that the reduced communication and faster aggregation of Parameter Box significantly affects runtime. In addition, we provide speedup when training with only 7 workers and Parameter Box, *so that the total machine count in the system is equal to the baseline.*

Figure 6.3 shows training performance on 56 Gbps InfiniBand. In this setup, for networks

such as GoogleNet, Inception, ResNet, and ResNext, forward and backward pass execution time limits training throughput; thus, raising network bandwidth only marginally affects the total throughput of Parameter Box versus MxNet. Since Parameter Hub never slows down training, we omit results of these networks (1x speedup) for clarity. We expect larger speedups with newer, faster GPUs such as the NVidia V100, for these networks. Significant speedup is still achieved with models that have large communication-to-computation ratios, such as AlexNet and VGG; these models remained network-bound even on 56 Gbps links.

The gap between PShard and MxNet signifies the benefit of software optimizations. The gap between PShard and Parameter Box highlights both the benefit of a non-colocated server that *halves the per link bandwidth usage, yielding a significant performance difference*, and the benefit of the hardware optimizations.

Figure 6.4 breaks down the overhead in different distributed training stages when running Parameter Hub in the same setup as Figure 3.1. Compared to Figure 3.1, *Parameter Hub reduces overheads from data copy, aggregation, optimization, and synchronization, and fully overlaps these stages, shifting the training back to a compute-bound workload.*

6.1.4 Performance with Infinitely Fast Compute

We used a benchmark to assess the efficiency of Parameter Hub's gradient processing pipeline to avoid being bottlenecked by our workers' GPUs. We implemented a special MxNet engine, called `ZeroComputeEngine`, based on the original `ThreadedEnginePerDevice`, which replaces training operators (such as convolution) with an empty routine. Only the synchronization operators (`WaitForVar`, `KVStoreDistPush` and `KVStoreDistPull`) are actually executed. This engine effectively simulates arbitrarily fast forward and backward passes on the worker side, pushing the limits of Parameter Hub.

We used ResNet 18 as the test network. We measured how fast each worker can run in this setup individually with the Parameter Box, then gradually added more workers to plot total system throughput.

Figure 6.5 shows the results of running the benchmark with Parameter Box, PShard and

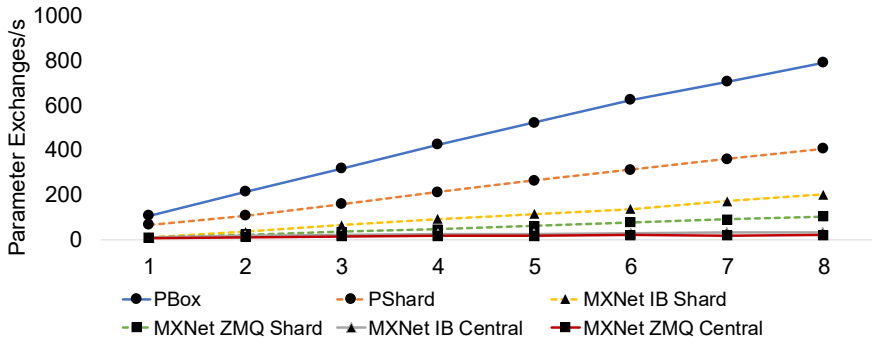

Figure 6.5: Parameter Box provides linear scaling of throughput for 8 worker nodes with infinitely fast compute, training ResNet 18.

multiple baseline configurations. Parameter Box provided linear scaling with 8 workers and outperformed the baseline by a large margin (up to 40x). Parameter Box had 2x the speedup of PShard because each of its interfaces needed to move only about half the amount of data compared to colocated servers.

6.1.5 Exploiting Locality

To postpone hitting the memory bandwidth limit, it is crucial to exploit locality in network interfaces and processor caches. This section evaluates the effectiveness of Parameter Hub's key assignment scheme and tall aggregation/optimization in leveraging locality.

Key Affinity in Parameter Box: We evaluate two schemes for connecting workers to Parameter Box to exploit locality and load balancing. In *Key by Interface/Core mode*, workers partition their keys equally across different interfaces on the Parameter Box. This mode better utilizes cache by binding a key to a particular interface, core and a NUMA node. This mode also exploits locality in time as workers are likely to generate the same key close to each other in synchronous training.

In *Worker by Interface mode*, each worker communicates with the server through a single interface. This lets Parameter Hub exploit locality within a single worker. It also provides naturally perfect load balancing across interfaces and cores at the cost of additional commu-

	Mem BW	Throughput
Opt/Agg Off	77.5	72.08
Caching Opt/Agg	83.5	71.6
Cache-bypassed Opt/Agg	119.7	40.48

Table 6.2: Bidirectional memory bandwidth (GB/s) utilization in Parameter Hub when training VGG with 8 workers. The maximum memory bandwidth for the machine is 137 GB/s for read-only workloads and 120 GB/s for 1:1 read:write workloads as measured by LikWid and Intel MLC.

nication and synchronization for each key within the server because keys are scattered across all interfaces and sockets.

We found that Key by Interface/Core provided 1.43x (790 vs 552 exchanges/s) better performance than Worker by Interface mode with `ZeroComputeEngine`. The locality within each worker could not compensate for synchronization and memory movement costs.

Tall vs. Wide Parallelism: We evaluated tall aggregation vs MxNet's wide approach with ResNet 50. Tall outperformed wide by 20x in terms of performance and provides near-perfect scaling to multiple cores. Tall aggregation benefited from increased overlap compared to wide, and wide was further hurt by the cost of synchronization.

Caching Effectiveness in Parameter Hub: Caching benefits many Parameter Hub operations. For example, models can be sent directly from cache after being updated, and aggregation buffers can reside in cache near the cores doing aggregation for those keys. We now evaluate the effectiveness of caching in Parameter Hub by measuring memory bandwidth usage.

Table 6.2 shows the memory bandwidth costs of communication, aggregation, and optimization on Parameter Box. We used 8 workers running a communication-only benchmark based on the VGG network, chosen because it had the largest model size. We first ran the benchmark with no aggregation or optimization, and we then added our two aggregation and optimization implementations.

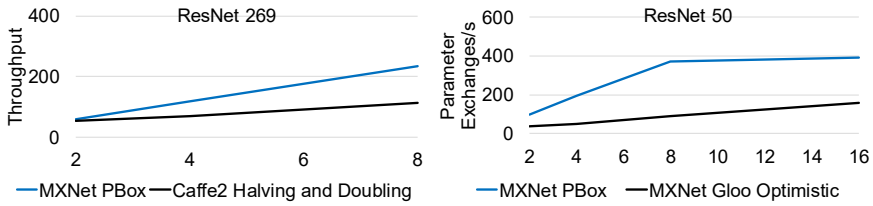

Figure 6.6: Left: Comparing Caffe2 + Gloo and MxNet + Parameter Box on an 10Gbps InfiniBand network. Right: Comparing MxNet + Gloo and MxNet + Parameter Box on a 56Gbps InfiniBand network with `ZeroComputeEngine`.

Without aggregation and optimization, Parameter Box's bidirectional memory bandwidth usage was stable at 77.5 GB/s. No cache was used in this case because Parameter Box did not touch the data (only the network interface did).

We found that the caching version of the aggregator and optimizer performed significantly better than the cache-bypassing version, which hit the maximum memory bandwidth available on the Parameter Hub machine when combined with the memory bandwidth of worker sends and receives. The caching version, on the other hand, added only 8% to total memory bandwidth usage; aggregation and optimization added only 1% of overhead to the overall throughput in this benchmark, fully overlapping gradient transfer.

6.1.6 Comparison with Other Schemes

Parameter servers are not the only way to perform model updates. Frameworks such as CNTK and Caffe2 can use HPC-like approaches, such as collective communication operations [144, 62].

To understand how Parameter Hub compares to other communication schemes, we first ran Caffe2 and MxNet with Parameter Box. We used InfiniBand for both systems. We evaluated the fastest algorithm in Gloo: recursive halving and doubling, used in [53]. Figure 6.6 (left) shows Parameter Box was nearly 2x faster.

We ported Gloo to MxNet to better assess both systems. Gloo implements blocking

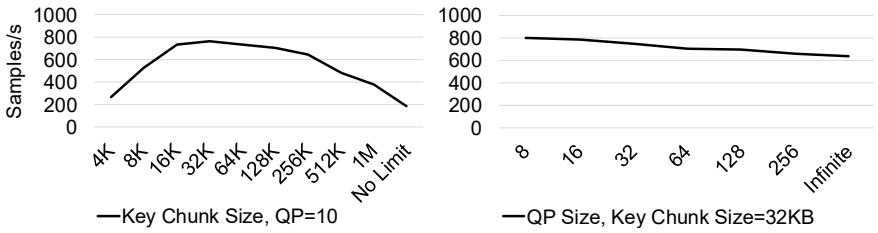

Figure 6.7: Effect of chunk size and queue pair count on throughput.

collective operations, but MxNet expects non-blocking operations. Therefore, we measured an optimistic upper bound by letting Gloo start aggregating the entire model as soon as the backward pass started, as if all gradients were available instantaneously. Since Gloo only does reduction, we ran our SGD/Nesterov optimizer on all nodes after reduction was complete. We used 56 Gbps IB and `ZeroComputeEngine` to compute bottlenecks. Figure 6.6 (right) shows Parameter Box sustained higher throughput and provided better scaling up to its limit. Two reasons account for this difference. First, collectives suffer from the same problem as colocated PSs: the interface on each participating node must process nearly 2x the data (Gloo's `allreduce` starts with a `reduce-scatter` followed by an `allgather` [144]). Second, collectives frequently use multi-round communication schemes whereas Parameter Box uses only 1 round.

6.1.7 Tradeoffs in Fine-Grained Key Chunking

We now examine tradeoffs in the communication layer concerning the size of key chunks and queue pair counts.

Size of key chunks: Parameter Hub leverages fine-grained key chunking to better balance load and overlap gradient reception and aggregation. Figure 6.7 (left) evaluates the effect of chunk size with `ZeroComputeEngine` on Parameter Box. Larger chunk sizes improved network utilization, while smaller sizes improved overlapping. We found 32KB chunk size to be optimum: this is likely due to our interfaces' maximum injection rate and aggregation pipeline latency.

Queue Pair Count: A worker needs at least one queue pair per interface with which it communicates. Queue pairs have state, which is cached on the card. When that cache misses frequently, communication slows. For Parameter Box to use 10 interfaces, we need a minimum of 10 queue pairs per worker. More queue pairs could enable concurrent transmission from the same worker and reduce head of line blocking, but it increases the queue pair cache miss rate. Figure 6.7 (right) evaluates the tradeoff, showing that fewer queue pairs was optimal.

6.1.8 Limits on Scalability

The scalability of Parameter Hub is inherently limited by available total memory, network or PCIe bandwidth. This section explores how close Parameter Hub gets to these limits. We use Parameter Box to answer these questions. Parameter Box achieves a 1:1 read:write memory bandwidth of 120 GB/s and a bidirectional network bandwidth of 140 GB/s. To determine how much bandwidth can be utilized, we added an additional IB interface to each of our 8 machines to emulate 16 workers and configured varying numbers of emulated workers running `ib_write_bw`, each with 10 QP connections to the `ib_write_bw` process on Parameter Box. These pairs of processes did repeated RDMA-writes to two 1 MB buffers on the other side. We set the PCIe read request size to 512 bytes. This configuration was chosen to mirror the setup of an actual training system while maximizing total system throughput.

To our surprise, we found that the peak memory bandwidth usage never exceeded more than 90 GB/s, far from the limit of both the aggregate network card and memory. This suggests that the bottleneck lies somewhere else.

We then built a loopback microbenchmark that used the IB cards to copy data locally between RDMA buffers. This isolated the system from network bottlenecks and involved only the NIC's DMA controllers and the processor's PCIe-to-memory-system bridge. This microbenchmark also achieved only 90 GB/s. Based on this experiment, we believe that *the limit of throughput in our current Parameter Hub system is the PCIe-to-memory-system bridge.*

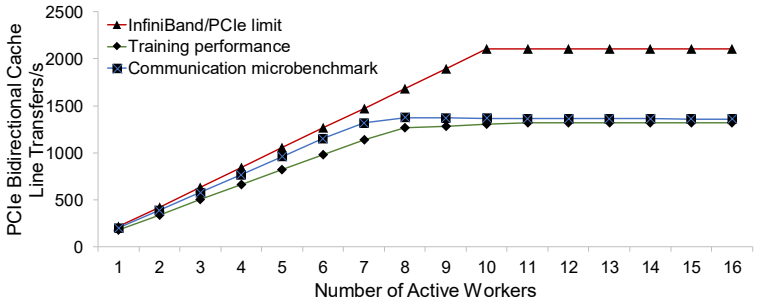

Figure 6.8: Parameter Box scalability is limited by the throughput of the PCIe to the on-chip network bridge of the PBox processors. Parameter Hub can utilize 97% of the measured peak PCIe bandwidth.

Figure 6.8 summarizes this experiment. The InfiniBand/PCIe limit line shows an ideal case where unlimited cache line transfers can be performed. However, this rate was not achievable even with a microbenchmark, which poses a hard upper bound on how fast Parameter Hub can run during training. We also see that, when training VGG with `ZeroComputeEngine`, as more workers are added, Parameter Box's performance approached the microbenchmarks (97%), demonstrating Parameter Hub's ability to fully utilize system resources. The gap in the plot between Parameter Box and the microbenchmark is due to the overhead of scheduling operations in MxNet and straggler effects in workers. Parameter Box hit the limit at a sustained 80GB/s memory bandwidth.

In real training, however, Parameter Box's scalability limit was difficult to reach. Recent work ([73, 78]) describes the difficulty of generalization with large batch sizes; it is not advantageous to blindly scale deep learning to a large number of workers without considering statistical efficiency [156, 76]. One example [53] reports that ResNet 50's statistical efficiency drops with aggregate batch sizes larger than 8192 on a system with 256 GPUs on 32 machines (with a mini-batch size of 32 per GPU). To assess whether Parameter Box could reach this scale, we measured the memory bandwidth usage for ResNet 50 with 8 workers using the same batch size. We found that Parameter Box required only 6GB/s memory bandwidth and an aggregated 4GB/s network bandwidth. This suggests that our Parameter Box prototype

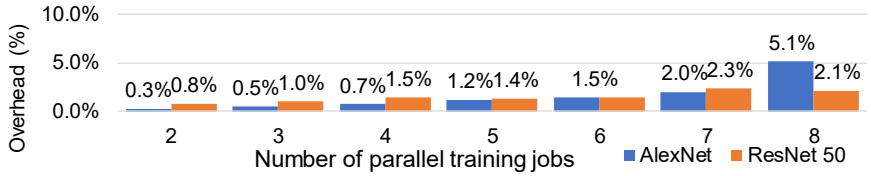

Figure 6.9: Overhead of multiple parallel training jobs sharing the same Parameter Box instance.

could scale to rack-level and support up to 120 worker machines training this network. In other words, our prototype could support sufficient scalability to handle cutting-edge training tasks.

On the other hand, the scalability bottleneck (PCIe controller) in our current prototype is specific to this particular platform, but it can change. For example, recently released AMD Epyc [11] processors provide nearly triple the Stream Triad performance (290 GB/s) [12] and 40% more PCIe bandwidth than our Parameter Box machine. We would expect Epyc to support 40% more throughput.

6.1.9 Effectiveness of Parameter Box as a Rack-Scale Service

We now evaluate effectiveness of Parameter Box as a rack-scale service with two typical scenarios in a 10 Gbps cloud-like environment: (1) when multiple jobs are training in parallel in a rack, sharing the same Parameter Box instance with different key namespaces and (2) when a training job crosses rack boundaries, and Parameter Hub performs hierarchical reduction.

Figure 6.9 shows the overhead of running multiple independent training jobs sharing a single Parameter Box. AlexNet saw a 5% drop in per-job throughput when running 8 jobs, likely due to frequent invocation of optimizer and less effective caching; ResNet 50 saw a smaller impact as it is compute bound.

Figure 6.10 emulates a single cloud-based training job whose VMs span N racks, and each rack contains 8 workers and 1 Parameter Box. The Parameter Box uses a widely used ring reduction algorithm [16, 130] for inter-rack aggregation.

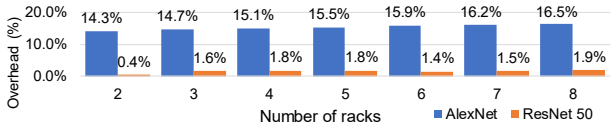

Figure 6.10: Emulated overhead of hierarchical reduction with Parameter Box.

Since we have only one Parameter Box machine, we model this ring reduction by sending and receiving N chunk-size messages sequentially, each performing one additional aggregation, for each of the keys, after local rack has finished aggregation. We assume each rack would finish its local aggregation at roughly the same time, as stragglers can exist regardless of rack assignment. Therefore, this faithfully estimates overhead of Parameter Hub's hierarchical reduction.

AlexNet's throughput loss comes from added latency of multiple rounds of communication, but is compensated by drastically reduced cross-rack traffic, and thus we would expect speedup in real deployment. On the other hand, we again observed virtually no loss of throughput in ResNet 50.

6.1.10 Rack-scale cost model

Is a cluster built with PHubs and a slow network more cost effective than one with sharded PSs and a fast network? This section explores this question using a simple cost model. We consider the cost of three cluster components: worker nodes, PHub nodes, and network gear. We use advertised prices from the Internet; while a datacenter operator might pay less, the ratios between component prices should still be similar. The baseline is a cluster running MXNet IB with colocated sharded PSs; we compare this to a PHub deployment in terms of throughput per dollar.

The model works by computing the cost of a worker node, and adding to it the amortized cost of its network usage; for the PHub deployment, it also includes the amortized cost of the worker's PHub usage. This allows us to compare the cost of worker nodes in deployments with different numbers of workers per rack, switch, or PHub. We capture only the most

significant cost, and include only capital cost, since operational costs are dominated by GPU power usage and thus differences would be small.

We model a standard three-layer datacenter network with some simplifying assumptions: racks hold as many machines as may be connected to a single switch, all switches and cables are identical, and oversubscription happens only at ToR switches. We model network costs by charging each worker the NIC per-port cost N, the amortized cost of one ToR switch port S and cable C, and fractional costs of ToR/aggregation/core switch ports and cables depending on the oversubscription factor F. Thus, the amortized cost of the network per machine is $A = (N + S + C) + F(4S + 2C)$. Since our goal is to model costs for future deployments, we make two changes from our experimental setup. Instead of 10Gb IB, we use 25 Gb Ethernet. Instead of NVIDIA 1080 Ti's, we assume a future GPU with similar cost G, but that performs like today's V100. This keeps the compute/communication ratio similar to that of our experiments. We use ResNet-50 for comparison; we use our 10Gb IB results for the PHub setting and downclocked 40Gb IB for the MXNet IB baseline. We include 2% overhead in the PHub numbers to account for aggregation between racks.

Workers are the same as in our evaluation, but with 4 GPUs. The cost W is \$4117 [142] without GPUs; the GPU price G is (\$699 [108]). The 100Gb baseline uses Mellanox ConnectX-4 EN cards (\$795 [107]) and 2m cables (\$94 [106]). The 25Gb PHub workers use Mellanox ConnectX-4 Lx EN cards (\$260 [107]) and 4-to-1 breakout cables (\$31.25 per port [106]). The PHub node (also same as evaluation) cost H is \$8407 [141], plus 10 dual-port 25Gb Mellanox ConnectX-4 Lx EN cards (\$162.5 per port [107]). The cost of each baseline worker is $W + N + 4G + A$, and the cost of a PHub worker is $W + N + 4G + A + KP$, where KP is the amortized PHub cost ($P = W + 20N + 20A$; K is the worker to Parameter Hub ratio).

We use the Arista 7060CX-32S 32-port 100Gb Ethernet switch (\$21077 [14]) in both configurations, with breakout cables to connect 25Gb hosts. With no oversubscription, each switch supports 16 100Gb baseline workers, or a PHub and 44 25Gb workers. With 2:1 oversubscription each switch could support a Parameter Hub and 65 25Gb workers; with 3:1, the number of supported workers is 76.

	Throughput/$1000		
	Future GPUs	Spendy	Cheap
100Gb Sharded 1:1	46.11	14.57	60.41
25Gb PHub 1:1	55.19	15.30	77.21
25Gb PHub 2:1	57.71	15.49	82.24
25Gb PHub 3:1	59.03	15.58	84.95

Table 6.3: Datacenter cost model comparing 25GbE PHub deployments with 100GbE MXNet IB on ResNet-50. Higher is better. The Future GPU PHub deployment with 2:1 oversubscription provides 25% better throughput per dollar.

Table 6.3 compares a full-bisection-bandwidth 100GbE sharded MXNet IB deployment with 25GbE PHub deployments with varying oversubscription. With 2:1 oversubscription, the PHub deployment provides 26% better throughput per dollar. We consider two other configurations: a "lower bound" using today's expensive V100's, where the 2:1 PHub deployment provides only 6% improvement; and a "GPU-focused" one using cheap CPUs (E5-2603 v4) in workers, providing 36% improvement.

6.2 Effectiveness of Parameter Link

We continue to evaluate Parameter Link in public cloud environments.

Our evaluation goals are as follows: (1) Measure Parameter Link's impact on end-to-end training. (2) Demonstrate the efficiency of AggEngine. (3) Quantify the benefit of hierarchical aggregation and topology-awareness. (4) Evaluate the accuracy of ProbeEmbed's inference of physical affinity. (5) Assess how well Autotune reacts to network changes.

6.2.1 Environment Setup

We perform our experiments on both Azure and EC2, in the same region or availability zone. Each VM runs Linux kernel 4.18 with SoftiWarp support [63], Cuda 9.2, CuDNN 7,

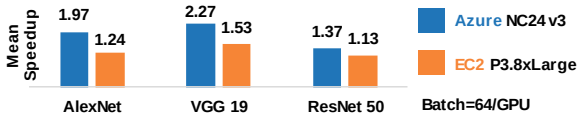

Figure 6.11: Parameter Link's collective optimizations achieve up to 2.27x speedup on public clouds training popular neural network models, compared to original Pytorch/Caffe2.

and network enhancements [94, 8] if possible. All VMs are provided with at least 10 Gbps network throughput, using DCTCP [4] as SoftRoCE and SoftiWarp do not yield better performance. AggEngine uses a chunk size tuned to its best performance, usually 16 to 64KB. All experiments use 64 nodes unless otherwise specified. Some experiments involve comparing results generated with uncertainty. In those cases, we report speedup in terms of mean and percentile metrics together with performance distribution of 50 runs. Each sample represents mean performance of 20 iterations using a potentially different cluster assignment generated by the underlying mechanism.

6.2.2 End to end training performance

We start our evaluation with the impact of Parameter Link's collective optimizations on the end-to-end training performance of popular neural networks. Then we breakdown the effect of each optimization in the following sections. We compare Parameter Link to original Pytorch/Caffe2 performance by replacing Gloo. We represent training speedup as mean speedup in throughput of 50 iterations to save experimental cost. This directly translates to a reduction in end-to-end training time as Parameter Link's optimization does not change convergence because it keeps the computation intact.

Figure 6.11 shows the speedup of Parameter Link, which ranges from 1.37-2.27x on Azure, and 1.13x-1.53x on EC2[1] when training popular vision models. We expect larger speedups for neural networks with higher communication to computation ratios (such as AlexNet and

[1]While we report only data-parallel results for its dominance in distributed training, Parameter Link optimizations are paradigm-agnostic, as all the scheduling of transfers is done by the framework.

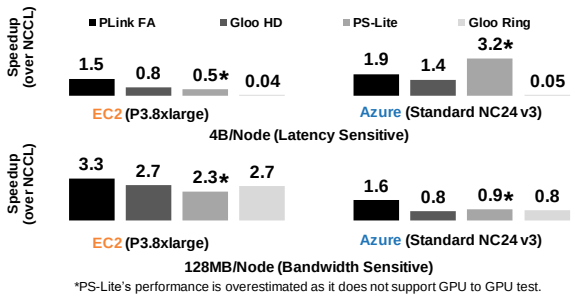

Figure 6.12: GPU to GPU aggregation speedup of various systems on Azure and EC2 in terms of mean latency for large and small buffers, normalized to NCCL's performance.

VGG) on VMs with faster networks, as we observe near line-rate network utilization when training them with Parameter Link. For models with smaller communication to computation ratios (such as ResNet-50), Parameter Link's speedup comes from its reduced reduction latency.

6.2.3 Efficiency of AggEngine

This section evaluates the performance of AggEngine itself, disabling ProbeEmbed and Autotune. For clarity, we first set up our baseline with communication libraries used in major training frameworks, including MxNet PS-Lite, Nvidia NCCL 2.4, and Facebook Gloo, covering ring (Gloo, NCCL), tree (NCCL), halving-doubling (Gloo) and parameter server (PS-Lite). We use each library's benchmark program to measure its performance. On both Azure and EC2, we use instances with V100 GPUs and test end-to-end reduction performance involving copying from/to a GPU.

Figure 6.12 shows the speedup in terms of mean (20 iterations) GPU to GPU reduction latency normalized to NCCL's result of aggregating a large (128MB) and a small (4B) buffer. To provide a fair comparison, we limit AggEngine to use a single connection per peer (in a typical scenario, multiple connections are used). Overall, AggEngine's flat aggregation (FA) leads the pack and is thus used as a strong baseline.

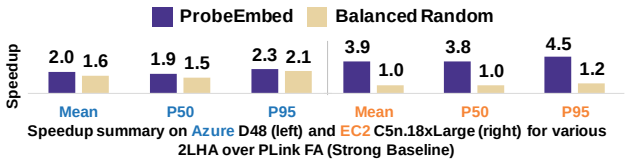

Figure 6.13: Mean, P50, and P95 additional speedup of 2LHA using different approaches over FA (strong baseline). Run on 64 Azure D64 series and EC2 C5n.18xLarge instances, with a constant VM topology throughout this experiment.

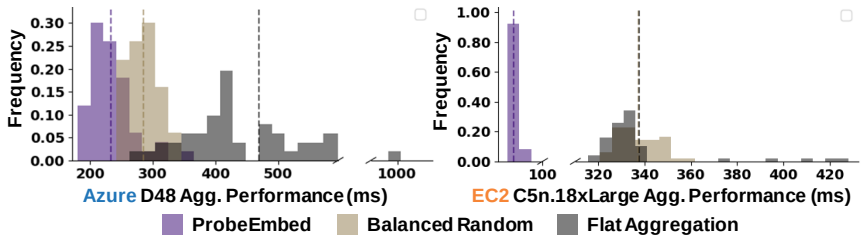

Figure 6.14: Empirical reduction latency distribution of ProbeEmbed-, Balanced Random- guided 2LHA and FA on Azure and EC2.

6.2.4 Effectiveness of Hierarchical Aggregation

We continue with a detailed comparison between Parameter Link FA and 2LHA, reducing a real-world model (ResNet-50, varying buffer sizes totalling ≈100MB). We use 4 connections per peer to fully utilize VM bandwidth. We separately tuned chunk sizes to ensure baseline perform well in both clouds. We present the speedup summary in Figure 6.13; then, each subsection details the benefits associated with the individual technique. Performance distribution is found in Figure 6.14.

Comparing with ground-truth-guided 2LHA. Cloud providers can expose topology information to assist in locality-aware scheduling and communication. We obtain ground-truth topology information from Azure. We look at the effectiveness of using physical topology on Azure to form a reduction schedule for 2LHA, grouping VM nodes based on their physical locality.

Our experiment shows that ProbeEmbed-guided 2LHA beats ground-truth-guided 2LHA

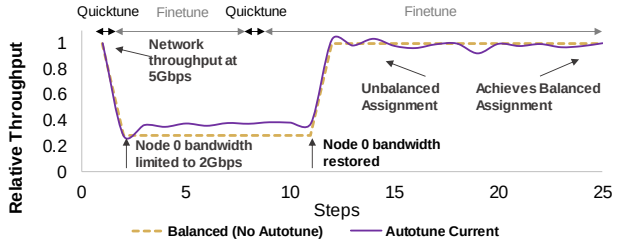

Figure 6.15: Autotune adapts to bandwidth changes by assigning LMs based on current metrics of AggEngine. Dotted lines: averaged throughput. Solid lines: snapshot throughput.

by 1.15x on average. The performance of ground-truth-guided 2LHA relies on a "luck" factor related to the physical topology of the VM nodes: the more compactly the VMs are allocated, and the more balanced the allocation in each rack is, the better performance of ground-truth-guided 2LHA should be. But VM spawn location is in total control of the scheduler, and sometimes it is impossible to guarantee a compact allocation due to current VM occupancy. It is difficult to splice/split undersized/oversized racks for a more balanced 2LHA without probing for network properties, which ProbeEmbed does.

Comparing with Balanced Random-guided 2LHA. We observe an additional 1.3x and 3.9x expected performance gain of the clusters generated by ProbeEmbed over those of Balanced Random on Azure and EC2. ProbeEmbed leads to better and more stable performance by generating cohesive, locality-preserving group assignments. On the other hand, Balanced Random can include VMs that are far away in the same group, creating a bottleneck in the system.

6.2.5 Accuracy of ProbeEmbed

Effective exploitation of locality requires capturing both static and dynamic aspects of the network (which is directly captured by the probes). In this section, we evaluate ProbeEmbed's ability to discover the network topology in terms of *physical affinity*.

Physical Affinity Inference Accuracy (PAIA). We define *affinity score* as an intuitive metric to quatify how well ProbeEmbed captures network topology: for any two VM nodes

(a, b) that have comparable distance to an observer c, let $T(a, c), T(b, c)$ be the ground-truth distance (in terms of hops) from a to c and b to c, and let $M(a, c), M(b, c)$ be the ProbeEmbed measured distance. We define an affinity score $A(a, b, c)$ for triplet (a, b, c) as:

$$A(a,b,c) = \begin{cases} 1 \textbf{ if } M(a,c) < M(b,c) \textbf{ and } T(a,c) < T(b,c) \\ \quad \textbf{or } M(a,c) > M(b,c) \textbf{ and } T(a,c) > T(b,c) \\ 0 \text{ otherwise} \end{cases}$$

For a given c, $A(a, b, c)$ captures whether ProbeEmbed's measurement agrees with the actual hop distance between a and b. $A(a, b, c)$ is only defined for *comparable* nodes a and b to c: each node's position is encoded as a tuple of format (region, datacenter, cluster, row, rack, host). a and b are comparable to c if they have different longest common prefix to c. Longer common prefix means higher affinity. We now define PAIA as the total sum of all $A(a, b, c)$ over the count of all defined $A(a, b, c)$s. A higher PAIA means a better comprehension of the datacenter network.

Quality of Various ProbeEmbed Probes. We evaluate base affinity accuracy achieved by running latency and bandwidth probes, without the embedding process, on 64 VMs, in 2 datacenters, 7 clusters, 15 rows and 61 racks in the US West 2 region with a total of 81.3K triplets to infer. Latency-based measurements better reflect underlying topology, with the DPDK Echo probe yielding a PAIA score of 95.6% compared to 77.4% with iperf, likely because bandwidth is not necessarily determined by distance.

Embedding's Effect on Affinity Inference. ProbeEmbed's probe readings can be affected by measurement noise. We now show how the embedding boosts ProbeEmbed's inference accuracy, in a case with 64 VMs in a single datacenter with 75.2K triplets. This effectively shrinks the latency distribution and makes it more difficult to infer topology. DPDK Echo probes without embedding yield a PAIA score of 81.3%. We apply embeddings with different α values to this dataset and found an average PAIA improvement of 5.0% with $\alpha = 1$, and 8.1% with $\alpha = 2$.

6.2.6 Effectiveness of Autotune

We conclude our evaluation with the real-world impact of Autotune on training ResNet-50 with Pytorch/Caffe2 on 8 nodes with FA. We report Autotune's effects in terms of end-to-end training performance. We run Autotune continuously, ignoring the stop condition, and assess how Autotune adapts to bandwidth changes and discovers a balanced LM assignment. We start by imposing no limits on bandwidth; then we limit the bandwidth of node 0, and eventually restore it. This shows how instantaneous training throughput changes as we apply Autotune decisions.

Figure 6.15 shows this process. We perform a step of Autotune every 10 iterations and report the snapshot reading of current throughput after the schedule change. At step=0, node 0's network throughput is ≈5 Gbps. At step=1, we limit its bandwidth to 2 Gbps. This causes an immediate drop in training performance and triggers Autotune at step=3. Quicktune moves LMs away from node 0. The training throughput then bumps up immediately, and Autotune enters fine-tuning mode through step=11. During this period, Autotune continues to move LM assignments away from 0. When 0 is out of LMs, Finetune moves LMs among the rest of the nodes based on their blame score. At step=11, node 0's bandwidth is restored, causing an immediate jump in training performance, but this time node 0 is underloaded. At step=12, this is partially corrected by Quicktune. Autotune continues to monitor and rebalance workloads throughout, arriving at a balanced assignment at t=25, with $\frac{max_n \sum_i D(n,i)}{min_n \sum_i D(n,i)} < 1.05$. Compared to using this near optimally balanced LM assignment alone, Autotune delivers 1.27x speedup when node 0 is the bottleneck, and can quickly recover when node 0's limit is lifted.

6.3 Evaluation

We evaluate ☁Collectives with a series of microbenchmarks from various communication backends and real-world applications that use collectives. We represent speedup by comparing the performance we get from the best rank order and the worst rank order. We avoid

Setup	Azure	EC2
Gloo Ring 100MB	0.58	0.78
OpenMPI Ring 100MB	0.81	0.94

Table 6.4: Spearman correlation coefficient between predicted performance from cost model and actual performance.

comparing with the original rank order because it is random, and has a wide performance distribution (Figure 5.1).

6.3.1 Experimental Setup

Our experiments are conducted on two public clouds, Azure and EC2. We enable network acceleration on both clouds and set TCP congestion control protocol to DCTCP. We include microbenchmarks that exercise ring, having doubling, double binary tree, and Bcube algorithms. All experiments run on Ubuntu 19. We focus our evaluation on one of the most important tasks in collectives, *allreduce* for its popularity and generality.

6.3.2 Prediction Accuracy of Cost Model

While the goal of the cost model is not to predict the actual performance, but rather, it should preserve the relative order of performance, i.e., $p_{pred}(\mathbb{R}_1) < p_{pred}(\mathbb{R}_2) \implies p_{real}(\mathbb{R}_1) < p_{real}(\mathbb{R}_2)$ should hold true for as many pairs of (R_i, R_j)s as possible. We demonstrate this for ring based collectives by generating 10 different rank orders, with the i-th order approximately corresponds to the $10i$-th percentile in the range of costs found by the solver. We obtain performance data for Facebook Gloo and OpenMPI running OSU Benchmark on 64 F16 nodes on Azure and 64 C5 nodes on EC2. We then compute Spearman [39] correlation coefficient between the predicted performance and the actual performance for each setup (Table 6.4).

6.3.3 Microbenchmark Performance

We evaluate ☁Collectives's efficacy with microbenchmarks of algorithms introduced in 5. We report a mean speedup of 20 iterations. We run all benchmarks with 512 F16 nodes on Azure, except for NCCL, which runs on 64 P3.8xLarge GPU nodes on EC2. Specifically, we set $B = 4$ for BCube; for NCCL, we use a single binary tree reduction for small buffers and a ring for large buffers. In all benchmarks, we reduce a buffer of 100MB, except for Nvidia NCCL, where we reduce a small buffer of 4B to trigger the tree algorithm.

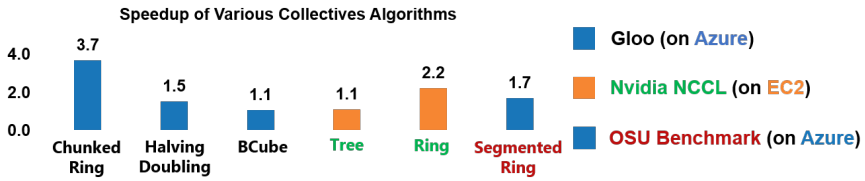

Figure 6.16: Speedups achieved by just using the rank ordering produced by our tool in various algorithms across multiple distributed communication backends at a large scale.

Figure 6.16 shows a summary of speedups achieved using ☁Collectives on these benchmarks, ranging from 1.1x-3.7x, with the ring-family algorithms benefiting the most. We speculate the reason for the effectiveness is that they have a much wider performance distribution, as each permutation of the order can potentially generate a different performance (cost of each hop is on the critical path); they also have simpler cost model, allowing the solvers to quickly navigate the objective landscape. On the other hand, halving doubling, BCube, and tree algorithms have complex objectives – sum of maximums, resulting in a narrower performance distribution because mutation of the ordering may not change the cost at all if the mutation does not cause the critical path to change.

6.3.4 End-to-end Performance Impact on Real-world Applications

Speedup Distributed Gradient Boosted Decision Tree Training. We evaluate ☁Collectives's impact on LightGBM [72], a gradient boosted decision tree training system. We use data

parallelism to run *lambdarank* with metric *ndcg*. Communication-wise, this workload runs two tasks: *allreduce* and *reducescatter*, and they are called sequentially in each split of the iteration. At our scale of 512 nodes, LightGBM automatically chooses to use halving and doubling for both *reducescatter* and *allreduce*. We use a dataset that represents an actual workload in our commercial setting with 5K columns and a total size of 10GB for each node. We train 1K trees, each with 120 leaves. We exclude the time it takes to load data from disk to memory and report an average speedup of 1000 iterations. ☁Collectives generated rank orders speed up training by 1.3x.

Speedup Distributed Deep Neural Network Training. We show ☁Collectives's effectiveness on distributed training of DNNs with Caffe2/Pytorch, on 64 EC2 p3.8xLarge nodes with data parallelism and a batch size of 64/GPU. We train AlexNet on the ImageNet dataset. Since our ☁Collectives does not change computation and only improves communication efficiency of the *allreduce* operation at iteration boundary, we report speedup of training in terms of images/second, averaged across 50 iterations. We use the ring chunked algorithm, which achieves the best baseline performance, and the ☁Collectives-optimized rank ordering of VM nodes achieves a speedup of 1.2x.

Chapter 7

CONCLUSION

The surge of data volume and accelerator throughput demand new optimizations in communication to accelerate distributed learning. This paper argues deficiency in communication comes from hardware, software and the network infrastructure itself. Through co-designing of software and hardware and careful probing and exploiting of locality in the physical network, we demonstrate significant end-to-end training speedup can be obtained.

BIBLIOGRAPHY

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pages 265–283, Savannah, GA, 2016. USENIX Association.

[2] Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. Extremely large minibatch SGD: training resnet-50 on imagenet in 15 minutes. CoRR, abs/1711.04325, 2017.

[3] M. Alfatafta, Z. AlSader, and S. Al-Kiswany. Cool: A cloud-optimized structure for mpi collective operations. In 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pages 746–753, July 2018.

[4] Mohammad Alizadeh, Albert Greenberg, Dave Maltz, Jitu Padhye, Parveen Pate, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In SIGCOMM 2010. ACM, September 2010.

[5] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In Proceedings of the ACM SIGCOMM 2010 Conference, SIGCOMM '10, page 63–74, New York, NY, USA, 2010. Association for Computing Machinery.

[6] Amazon. Amazon ec2 f1 instances. `https://aws.amazon.com/ec2/instance-types/f1`. (Accessed on 03/04/2019).

[7] Amazon. Deep learning on aws. `https://aws.amazon.com/deep-learning/`. (Accessed on 12/06/2018).

[8] Amazon. Enable and configure enhanced networking for ec2 instances. `https://aws.amazon.com/premiumsupport/knowledge-center/enable-configure-enhanced-networking/`. (Accessed on 01/06/2019).

[9] Amazon. New c5n instances with 100 gbps networking | aws news blog. https://aws.amazon.com/blogs/aws/new-c5n-instances-with-100-gbps-networking/. (Accessed on 02/04/2020).

[10] Amazon. Placement groups - amazon elastic compute cloud. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/placement-groups.html. (Accessed on 12/07/2018).

[11] AMD. AMD EPYC. `http://www.amd.com/en/products/epyc`.

[12] AMD. Epyc benchmarks. `https://www.amd.com/en/products/epyc-benchmarks`.

[13] Garth Gibson Alexandra Fedorova Gennady Pekhimenko Anand Jayarajan, Jinliang Wei. Priority-based parameter propagation for distributed dnn training. 2019.

[14] Arista. Arista 7060cx-32s price. `https://goo.gl/cqyBtA`.

[15] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007.

[16] Baidu. baidu-research/baidu-allreduce. `https://github.com/baidu-research/baidu-allreduce`. (Accessed on 05/14/2018).

[17] Vasanth Bala, Jehoshua Bruck, Robert Cypher, Pablo Elustondo, Alex Ho, Ching-Tien Ho, Shlomo Kipnis, and Marc Snir. Ccl: A portable and tunable collective communication library for scalable parallel computers. IEEE Transactions on Parallel and Distributed Systems, 6(2):154–164, 1995.

[18] Dimitris Bertsimas and John Tsitsiklis. Simulated annealing. Statistical Science, 8, 02 1993.

[19] Kashif Bilal, Samee Ullah Khan, Joanna Kolodziej, Limin Zhang, Khizar Hayat, Sajjad Ahmad Madani, Nasro Min-Allah, Lizhe Wang, and Dan Chen. A comparative study of data center network architectures. In ECMS, 2012.

[20] Edward K Blum, Xin Wang, and Patrick Leung. Architectures and message-passing algorithms for cluster computing: Design and performance. Parallel Computing, 26(2-3):313–332, 2000.

[21] Stephen Boyd and Almir Mutapcic. Subgradient methods. lecture notes of EE392o, Stanford University, Autumn Quarter, 2004, 01 2003.

[22] PS Bradley, KP Bennett, and Ayhan Demiriz. Constrained k-means clustering. Microsoft Research, Redmond, 20(0):0, 2000.

[23] D. Bureddy, H. Wang, A. Venkatesh, S. Potluri, and D. K. Panda. Omb-gpu: A micro-benchmark suite for evaluating mpi libraries on gpu clusters. In Jesper Larsson Träff, Siegfried Benkner, and Jack J. Dongarra, editors, Recent Advances in the Message Passing Interface, pages 110–120, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[24] Martin Burtscher and Paruj Ratanaworabhan. Fpc: A high-speed compressor for double-precision floating-point data. IEEE Transactions on Computers, 58(1):18–31, 2009.

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

[26] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. arXiv preprint arXiv:1604.06174, 2016.

[27] 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, 2014. USENIX Association.

[28] Minsik Cho, Ulrich Finkler, and David Kung. Blueconnect: Novel hierarchical all-reduce on multi-tired network for deep learning. In LearningSys 2018, 2018.

[29] Francois Chollet. Xception: Deep learning with depthwise separable convolutions. 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Jul 2017.

[30] Cisco. The market need for 40 gigabit ethernet. `http://www.cisco.com/c/en/us/products/collateral/switches/catalyst-6500-series-switches/white_paper_c11-696667.html`.

[31] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R Ganger, Phillip B Gibbons, et al. Exploiting bounded staleness to speed up big data analytics. In USENIX Annual Technical Conference, pages 37–48, 2014.

[32] Henggang Cui, Hao Zhang, Gregory R. Ganger, Phillip B. Gibbons, and Eric P. Xing. GeePS: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In Proceedings of the Eleventh European Conference on Computer Systems, EuroSys '16, pages 4:1–4:16, New York, NY, USA, 2016. ACM.

[33] Henggang Cui, Hao Zhang, Gregory R Ganger, Phillip B Gibbons, and Eric P Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In Proceedings of the Eleventh European Conference on Computer Systems, page 4. ACM, 2016.

[34] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: A decentralized network coordinate system. SIGCOMM Comput. Commun. Rev., 34(4):15–26, August 2004.

[35] Wei Dai, Abhimanu Kumar, Jinliang Wei, Qirong Ho, Garth A. Gibson, and Eric P. Xing. High-performance distributed ML at scale through parameter server consistency models. CoRR, abs/1410.8043, 2014.

[36] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In International conference on Tools and Algorithms for the Construction and Analysis of Systems, pages 337–340. Springer, 2008.

[37] 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 Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1, NIPS'12, pages 1223–1231, USA, 2012. Curran Associates Inc.

[38] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018.

[39] Yadolah Dodge. Spearman Rank Correlation Coefficient, pages 502–505. Springer New York, New York, 2008.

[40] dpdk. Home - dpdk. `https://www.dpdk.org/`. (Accessed on 09/01/2019).

[41] Mixpanel Engineering. 27 | october | 2011 | mixpanel engineering. `https://engineering.mixpanel.com/2011/10/27/`. (Accessed on 12/07/2018).

[42] Facebook. Distributed training | caffe2. `https://caffe2.ai/docs/distributed-training.html`. (Accessed on 05/09/2018).

[43] Facebook. facebookincubator/gloo: Collective communications library with various primitives for multi-machine training. `https://github.com/facebookincubator/gloo`. (Accessed on 12/22/2018).

[44] Facebook. gloo/algorithms.md at master · facebookincubator/gloo. https://github.com/facebookincubator/gloo/blob/master/docs/algorithms.md. (Accessed on 01/26/2020).

[45] Benjamin Farley, Ari Juels, Venkatanathan Varadarajan, Thomas Ristenpart, Kevin D. Bowers, and Michael M. Swift. More for your money: Exploiting performance heterogeneity in public clouds. In Proceedings of the Third ACM Symposium on Cloud Computing, SoCC '12, pages 20:1–20:14, New York, NY, USA, 2012. ACM.

[46] FastAI. Now anyone can train imagenet in 18 minutes · fast.ai. https://www.fast.ai/2018/08/10/fastai-diu-imagenet/. (Accessed on 12/20/2018).

[47] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. A configurable cloud-scale dnn processor for real-time ai. In Proceedings of the 45th Annual International Symposium on Computer Architecture, pages 1–14. IEEE Press, 2018.

[48] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. Annals of statistics, pages 1189–1232, 2001.

[49] Jinkun Geng, Dan Li, Yang Cheng, Shuai Wang, and Junfeng Li. Hips: Hierarchical parameter synchronization in large-scale distributed machine learning. In Proceedings of the 2018 Workshop on Network Meets AI and ML, NetAI'18, pages 1–7, New York, NY, USA, 2018. ACM.

[50] Y. Gong, B. He, and J. Zhong. Network performance aware mpi collective communication operations in the cloud. IEEE Transactions on Parallel and Distributed Systems, 26(11):3079–3089, Nov 2015.

[51] Google. Google cloud training. google cloud. https://cloud.google.com/training/courses/machine-learning-tensorflow-gcp. (Accessed on 03/04/2019).

[52] Google. Or-tools | google developers. https://developers.google.com/optimization. (Accessed on 01/26/2020).

[53] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: Training ImageNet in 1 hour. arXiv preprint arXiv:1706.02677, 2017.

[54] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, , Parantap Lahiri, Dave Maltz, and and. Vl2: A scalable and flexible data center network. Association for Computing Machinery, Inc., August 2009. Recognized as one of "the most important research results published in CS in recent years".

[55] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. ACM SIGOPS operating systems review, 42(5):64–74, 2008.

[56] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training. arXiv preprint arXiv:1806.03377, 2018.

[57] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H Campbell. Tictac: Accelerating distributed deep learning with communication scheduling. arXiv preprint arXiv:1803.03288, 2018.

[58] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[59] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More effective distributed ML via a stale synchronous parallel parameter server. In Advances in neural information processing systems, pages 1223–1231, 2013.

[60] Ningning Hu and Peter Steenkiste. Estimating available bandwidth using packet pair probing. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 2002.

[61] Ningning Hu and Peter Steenkiste. Evaluation and characterization of available bandwidth probing techniques. IEEE journal on Selected Areas in Communications, 21(6):879–894, 2003.

[62] Forrest N Iandola, Matthew W Moskewicz, Khalid Ashraf, and Kurt Keutzer. Firecaffe: near-linear acceleration of deep neural network training on compute clusters. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 2592–2600, 2016.

[63] IBM. Softiwarp for linux-rdma. `https://github.com/zrlio/softiwarp-for-linux-rdma`. (Accessed on 11/06/2019).

[64] Alexandru Iosup, Nezih Yigitbasi, and Dick Epema. On the performance variability of production cloud services. In Proceedings of the 2011 11th IEEE/ACM International

Symposium on Cluster, Cloud and Grid Computing, CCGRID '11, pages 104–113, Washington, DC, USA, 2011. IEEE Computer Society.

[65] iperf. iperf - the tcp, udp and sctp network bandwidth measurement tool. `https://iperf.fr/`. (Accessed on 09/01/2019).

[66] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based parameter propagation for distributed dnn training. arXiv preprint arXiv:1905.03960, 2019.

[67] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, and Changhoon Kim. Eyeq: Practical network performance isolation for the multi-tenant cloud. In Presented as part of the. USENIX, Submitted.

[68] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, Tiegang Chen, Guangxiao Hu, Shaohuai Shi, and Xiaowen Chu. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes, 2018.

[69] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. arXiv preprint arXiv:1807.05358, 2018.

[70] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA '17, pages 1–12, New York, NY, USA, 2017. ACM.

[71] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance RDMA systems. In 2016 USENIX Annual Technical Conference (USENIX ATC 16), pages 437–450, Denver, CO, 2016. USENIX Association.

[72] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In NIPS, 2017.

[73] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. arXiv preprint arXiv:1609.04836, 2016.

[74] Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun. Parallax: Sparsity-aware data parallel training of deep neural networks. In Proceedings of the Fourteenth EuroSys Conference 2019, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[75] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980, 2014.

[76] Alexandros Koliousis, Pijika Watcharapichat, Matthias Weidlich, Paolo Costa, and Peter Pietzuch. Crossbow: Scaling deep learning on multi-gpu servers.

[77] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems, pages 1097–1105, 2012.

[78] Y LeCun, L Bottou, and G Orr. Efficient backprop in neural networks: Tricks of the trade. Lecture Notes in Computer Science, 1524.

[79] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI'14, pages 583–598, Berkeley, CA, USA, 2014. USENIX Association.

[80] Mu Li, David G. Andersen, Alexander Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. In Proceedings of the 27th International Conference on Neural Information Processing Systems, NIPS'14, pages 19–27, Cambridge, MA, USA, 2014. MIT Press.

[81] Hyeontaek Lim, David G Andersen, and Michael Kaminsky. 3lc: Lightweight and effective traffic compression for distributed machine learning. arXiv preprint arXiv:1802.07389, 2018.

[82] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. arXiv preprint arXiv:1712.01887, 2017.

[83] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. IncBricks: Toward in-network computation with an in-network cache. SIGOPS Oper. Syst. Rev., 51(2):795–809, April 2017.

[84] Liang Luo, Ming Liu, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. Motivating in-network aggregation for distributed deep neural network training. 2017.

[85] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. Parameter hub: a rack-scale parameter server for distributed deep neural network training. In Proceedings of the ACM Symposium on Cloud Computing, pages 41–54. ACM, 2018.

[86] Liang Luo, Jacob S Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. Parameter hub: High performance parameter servers for efficient distributed deep neural network training. SysML, abs/1801.09805, 2018.

[87] Gustavo Marfia, Claudio Palazzi, Giovanni Pau, Mario Gerla, M. Y. Sanadidi, and Marco Roccetti. Tcp libra: Exploring rtt-fairness for tcp. In Ian F. Akyildiz, Raghupathy Sivakumar, Eylem Ekici, Jaudelice Cavalcante de Oliveira, and Janise McNair, editors, NETWORKING 2007. Ad Hoc and Sensor Networks, Wireless Networks, Next Generation Internet, pages 1005–1013, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[88] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. Taming performance variability. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pages 409–425, Carlsbad, CA, 2018. USENIX Association.

[89] Matt Mathis, John Heffner, and Raghu Reddy. Web100: extended tcp instrumentation for research, education and diagnosis. ACM SIGCOMM Computer Communication Review, 33(3):69–79, 2003.

[90] Matthew Mathis, Jeffrey Semke, Jamshid Mahdavi, and Teunis Ott. The macroscopic behavior of the tcp congestion avoidance algorithm. ACM SIGCOMM Computer Communication Review, 27(3):67–82, 1997.

[91] Qi Meng, Guolin Ke, Taifeng Wang, Wei Chen, Qiwei Ye, Zhi-Ming Ma, and Tie-Yan Liu. A communication-efficient parallel algorithm for decision tree. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, Advances in Neural Information Processing Systems 29, pages 1279–1287. Curran Associates, Inc., 2016.

[92] Bernard Metzler, Philip Frey, and Animesh Trivedi. Softiwarp – project update: A software iwarp driver for openfabrics. 03 2010.

[93] Microsoft. Azure Windows VM sizes - HPC. `https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes-hpc`. (Accessed on 01/11/2018).

[94] Microsoft. Create an azure virtual machine with accelerated networking | microsoft docs. `https://docs.microsoft.com/en-us/azure/virtual-network/create-vm-accelerated-networking-cli`. (Accessed on 01/06/2019).

[95] Microsoft. Introducing the new hb and hc azure vm sizes for hpc | azure blog and updates | microsoft azure. https://azure.microsoft.com/en-us/blog/introducing-the-new-hb-and-hc-azure-vm-sizes-for-hpc/. (Accessed on 02/04/2020).

[96] Microsoft. Machine learning studio. microsoft azure. `https://azure.microsoft.com/en-us/services/machine-learning-studio/`. (Accessed on 12/06/2018).

[97] Microsoft. Turing-nlg: A 17-billion-parameter language model by microsoft - microsoft research. `https://www.microsoft.com/en-us/research/blog/turing-nlg-a-17-billion-parameter-language-model-by-microsoft/`. (Accessed on 02/10/2020).

[98] Hiroaki Mikami, Hisahiro Suganuma, Pongsakorn U.-Chupala, Yoshiki Tanaka, and Yuichi Kageyama. Imagenet/resnet-50 training in 224 seconds. CoRR, abs/1811.05233, 2018.

[99] David Kung Minsik Cho, Ulrich Finkler. Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. 2019.

[100] Sándor Molnár, Balázs Sonkoly, and Tuan Trinh. A comprehensive tcp fairness analysis in high speed networks. Computer Communications, 32:1460–1484, 08 2009.

[101] MxNet. dmlc/ps-lite: A lightweight parameter server interface. https://github.com/dmlc/ps-lite. (Accessed on 12/22/2018).

[102] MxNet. Mxnet on the cloud — mxnet documentation. https://mxnet.incubator.apache.org/faq/cloud.html?highlight=ec2. (Accessed on 05/09/2018).

[103] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. Portland: a scalable fault-tolerant layer 2 data center network fabric. In SIGCOMM, 2009.

[104] Yurii Nesterov. A method for unconstrained convex minimization problem with the rate of convergence o (1/k2). In Doklady an SSSR, volume 269, pages 543–547, 1983.

[105] Nvidia. Ai research & development | nvidia dgx systems. https://www.nvidia.com/en-us/data-center/dgx-systems/. (Accessed on 02/05/2020).

[106] Nvidia. Mellanox ethernet cable prices. https://store.mellanox.com/categories/interconnect/ethernet-cables/direct-attach-copper-cables.html.

[107] Nvidia. Mellanox ethernet card prices. https://store.mellanox.com/categories/adapters/ethernet-adapter-cards.html.

[108] Nvidia. Nvidia 1080 ti advertised price. https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1080-ti.

[109] Nvidia. Nvidia collective communications library. https://developer.nvidia.com/nccl.

[110] Nvidia. Operations - nccl 2.3.4 documentation. https://docs.nvidia.com/deeplearning/sdk/nccl-developer-guide/docs/usage/operations.html. (Accessed on 12/07/2018).

[111] OpenAI. Ai and compute. https://openai.com/blog/ai-and-compute. (Accessed on 03/28/2019).

[112] OpenAI. Better language models and their implications. (Accessed on 03/28/2019).

[113] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. Journal of Parallel and Distributed Computing, 69(2):117–124, 2009.

[114] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In Proceedings of the 27th ACM Symposium on Operating

Systems Principles, SOSP '19, page 16–29, New York, NY, USA, 2019. Association for Computing Machinery.

[115] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. Improving network connection locality on multicore systems. In Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys '12, pages 337–350, New York, NY, USA, 2012. ACM.

[116] Aurick Qiao, Abutalib Aghayev, Weiren Yu, Haoyang Chen, Qirong Ho, Garth A. Gibson, and Eric P. Xing. Litz: Elastic framework for high-performance distributed machine learning. In 2018 USENIX Annual Technical Conference (USENIX ATC 18), pages 631–644, Boston, MA, 2018. USENIX Association.

[117] Rolf Rabenseifner. Optimization of collective reduction operations. pages 1–9, 06 2004.

[118] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In Advances in neural information processing systems, pages 693–701, 2011.

[119] Reddit. Openai: Better language models and their implications: Machine learning. (Accessed on 03/28/2019).

[120] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. Computer Communication Review, 45:123–137, 2015.

[121] Sebastian Ruder. An overview of gradient descent optimization algorithms. CoRR, abs/1609.04747, 2016.

[122] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Neurocomputing: Foundations of research. pages 696–699, Cambridge, MA, USA, 1988. MIT Press.

[123] Paul D. Sack. Scalable Collective Message-passing Algorithms. PhD thesis, Champaign, IL, USA, 2011. AAI3503864.

[124] Peter Sanders, Jochen Speck, and Jesper Larsson Träff. Two-tree algorithms for full bandwidth broadcast, reduction and scan. Parallel Comput., 35(12):581–594, December 2009.

[125] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter Richtárik. Scaling distributed machine learning with in-network aggregation, 2019.

[126] SciKit. 4.8. pairwise metrics, affinities and kernels. scikit-learn 0.20.2 documentation. `https://scikit-learn.org/stable/modules/metrics.html`. (Accessed on 02/06/2019).

[127] George AF Seber and Alan J Lee. Linear regression analysis, volume 329. John Wiley & Sons, 2012.

[128] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and application to data-parallel distributed training of speech DNNs. In Interspeech 2014, September 2014.

[129] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. CoRR, abs/1802.05799, 2018.

[130] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. CoRR, abs/1802.05799, 2018.

[131] Haichen Shen, Yuchen Jin, Bingyu Kong, M. Philipose, A. KrishnaMurthy, and Ravi Sundaram. Nexus : A gpu cluster for accelerating neural networks for video analysis. 2018.

[132] Alan Shieh, Srikanth Kandula, Albert Greenberg, and Changhoon Kim. Seawall: Performance isolation for cloud datacenter networks. In Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10, pages 1–1, Berkeley, CA, USA, 2010. USENIX Association.

[133] Alan Shieh, Srikanth Kandula, Albert Greenberg, Changhoon Kim, and Bikas Saha. Sharing the data center network. In Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation, NSDI'11, pages 309–322, Berkeley, CA, USA, 2011. USENIX Association.

[134] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. nature, 529(7587):484, 2016.

[135] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556, 2014.

[136] Samuel L. Smith, Pieter-Jan Kindermans, and Quoc V. Le. Don't decay the learning rate, increase the batch size. CoRR, abs/1711.00489, 2017.

[137] Alexander Smola and Shravan Narayanamurthy. An architecture for parallel topic models. Proc. VLDB Endow., 3(1-2):703–710, September 2010.

[138] SoftRoCE. Soft rdma over ethernet (roce) driver. `https://github.com/SoftRoCE/rxe-dev/wiki/rxe-dev:-Home`. (Accessed on 12/16/2018).

[139] Srinivas Sridharan, Karthikeyan Vaidyanathan, Dhiraj Kalamkar, Dipankar Das, Mikhail E. Smorkalov, Mikhail Shiryaev, Dheevatsa Mudigere, Naveen Mellempudi, Sasikanth Avancha, Bharat Kaul, and Pradeep Dubey. On scale-out deep learning training for cloud and hpc, 2018.

[140] Peng Sun, Wansen Feng, Ruobing Han, Shengen Yan, and Yonggang Wen. Optimizing network performance for distributed dnn training on gpu clusters: Imagenet/alexnet training in 1.5 minutes. arXiv preprint arXiv:1902.06855, 2019.

[141] SuperMicro. Supermicro phub node price. `https://www.thinkmate.com/system/superserver-6038r-txr`.

[142] SuperMicro. Supermicro worker node price. `https://www.thinkmate.com/system/superserver-1028gq-tr`.

[143] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Jun 2015.

[144] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. Int. J. High Perform. Comput. Appl., 19(1):49–66, February 2005.

[145] Nguyen Xuan Vinh, Julien Epps, and James Bailey. Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance. Journal of Machine Learning Research, 11(Oct):2837–2854, 2010.

[146] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Jorgen Thelin, Nikhil Devanur, and Ion Stoica. Blink: Fast and generic collectives for distributed ml, 2019.

[147] Jianyu Wang and Gauri Joshi. Adaptive communication strategies to achieve the best error-runtime trade-off in local-update sgd, 2018.

[148] Lipo Wang. Support vector machines: theory and applications, volume 177. Springer Science & Business Media, 2005.

[149] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Managed communication and consistency for fast data-parallel iterative analytics. In Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC '15, pages 381–394, New York, NY, USA, 2015. ACM.

[150] Wikipedia. Netlink wikipedia. `https://en.wikipedia.org/wiki/Netlink#cite_note-4`. (Accessed on 12/22/2018).

[151] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pages 595–610, Carlsbad, CA, 2018. USENIX Association.

[152] Pengtao Xie, Jin Kyu Kim, Qirong Ho, Yaoliang Yu, and Eric Xing. Orpheus: Efficient distributed machine learning via system and algorithm co-design. In Proceedings of the ACM Symposium on Cloud Computing, pages 1–13. ACM, 2018.

[153] Pengtao Xie, Jin Kyu Kim, Yi Zhou, Qirong Ho, Abhimanu Kumar, Yaoliang Yu, and Eric Xing. Distributed machine learning via sufficient factor broadcasting. arXiv preprint arXiv:1511.08486, 2015.

[154] Yang You, Igor Gitman, and Boris Ginsburg. Large batch training of convolutional networks. arXiv preprint arXiv:1708.03888, 2017.

[155] 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. In International Conference on Learning Representations, 2019.

[156] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. Speeding up imagenet training on supercomputers.

[157] 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, pages 1:1–1:10, New York, NY, USA, 2018. ACM.

[158] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In European conference on computer vision, pages 818–833. Springer, 2014.

[159] B. Zhang, Y. Ruan, and J. Qiu. Harp: Collective communication on hadoop. In 2015 IEEE International Conference on Cloud Engineering, pages 228–233, 2015.

[160] Ce Zhang and Christopher Ré. Dimmwitted: A study of main-memory statistical analytics. Proc. VLDB Endow., 7(12):1283–1294, August 2014.

[161] 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, 2017. USENIX Association.

[162] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning, 2016.