Parameter Hub: a Rack-Scale Parameter Server for Distributed Deep Neural Network Training

Liang Luo*, Jacob Nelson†, Luis Ceze*,
Amar Phanishayee†, Arvind Krishnamurthy*
*University of Washington, †Microsoft Research

Abstract

Distributed deep neural network (DDNN) training constitutes an increasingly important workload that frequently runs in the cloud. Larger DNN models and faster compute engines are shifting DDNN training bottlenecks from computation to communication. This paper characterizes DDNN training to precisely pinpoint these bottlenecks. We found that timely training requires high performance parameter servers (PSs) with optimized network stacks and gradient processing pipelines, as well as server and network hardware with balanced computation and communication resources. We therefore propose PHub, a high performance multi-tenant, rack-scale PS design. PHub co-designs the PS software and hardware to accelerate rack-level and hierarchical cross-rack parameter exchange, with an API compatible with many DDNN training frameworks. PHub provides a performance improvement of up to 2.7x compared to state-of-the-art cloud-based distributed training techniques for image classification workloads, with 25% better throughput per dollar.

1 Introduction

Most work in the systems and architecture community has focused on improving the efficiency of evaluating trained models. However, arriving at a trained model requires multiple lengthy experiments. Accelerating this training process lets developers iterate faster and design better models.

As DNNs get computationally more expensive to train, timely training requires exploiting parallelism with a distributed system, especially in the cloud [2, 6, 10]. The most common way of exploiting parallelism, "data" parallelism, consists of a computation-heavy forward and backward phase and a communication-heavy parameter exchange phase.

In this paper, we begin by performing a detailed bottleneck analysis of DDNN training and observe that the emergence of speedier accelerators shifts *the performance bottleneck of distributed DNN training from computation to communication*, because of the following factors.

First, the throughput of GPUs on a recent DNN, ResNet, has increased by 35x since 2012 on modern cloud-based GPUs (Figure 1), effectively demanding a similar increase in network bandwidth. Upgrading datacenter networks is expensive: compute instance network bandwidth on major cloud providers such as EC2 has improved little across generational upgrades [8], so care must be taken when configuring racks for DDNN training for optimal cost-efficiency.

Second, existing parameter exchange mechanisms such as parameter servers (PS) do not scale up the total throughput on a standard cloud network stack (Table 1) due to unoptimized software and hardware configurations, and lack of awareness of the underlying physical network topology.

The compound effect of these factors dramatically increases communication overhead during distributed DNN training. To illustrate this problem, Figure 2 shows a modest-scale DNN training with 8 machines on EC2 with 10 Gbps links[1]: modern DNN training frameworks can no longer hide the latency of communication due to faster computation. Spending most of the DDNN training time on model updates limits the benefit of faster GPUs.

Scaling cloud-based DDNN training throughput demands both a fast and a cost-effective solution. Our bottleneck findings show that such a solution requires a more optimized software stack, a specialized hardware design, and a more effective cluster configuration.

We therefore propose PHub, a high performance, multi-tenant, rack-scale PS design for cloud-based DDNN training. By co-designing the PS software with the hardware and the datacenter cluster rack configuration, PHub achieves up to 2.7x faster training throughput, with 25% better throughput per dollar. Our contributions include:

[1] Batch size per GPU (4, 16, 32, 32, saturating GRID 520) for each network is kept the same across all GPUs for easier comparison.

SoCC '18, October 11–13, 2018, Carlsbad, CA, USA

ACM ISBN 978-1-4503-6011-1/18/10…$15.00
https://doi.org/10.1145/3267809.3267840

Liang Luo*, Jacob Nelson†, Luis Ceze*,
Amar Phanishayee†, Arvind Krishnamurthy*

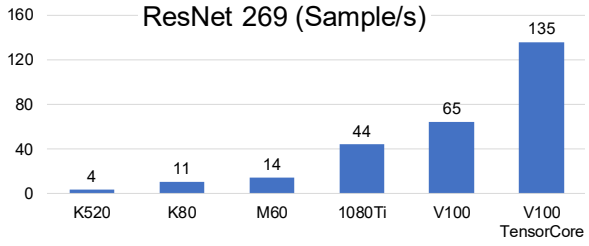


Figure 1: Single-GPU training throughput for ResNet 269 measured with MXNet on EC2 g2, p2, g3 and p3 instances, and a local GTX 1080 Ti machine, while maximizing GPU memory utilization. Per chip GPU throughput on ResNet 269 in the cloud has increased 35x since 2012.

Figure 2: The overhead of distributed training gets larger as GPUs get faster. The framework can no longer hide communication latency, and faster GPUs no longer improve training throughput. With today's fast GPUs, distributed cloud DNN training time is chiefly spent waiting for parameter exchanges.

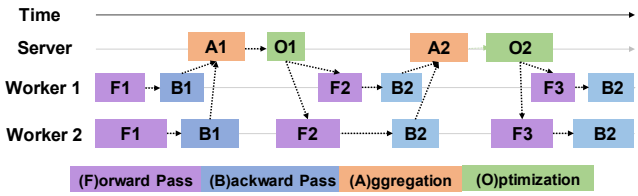
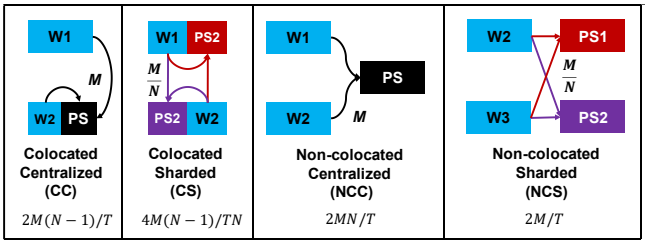


Figure 3: A few iterations in distributed training.

(1) A detailed bottleneck analysis of current state-of-the-art cloud-based DDNN training (§2).
(2) Design and implementation of the PHub PS software, supporting many DNN training frameworks (§3).
(3) A *balanced* central PS hardware architecture, PBox (§3.3), to leverage PHub for rack-level and hierarchical cross-rack gradient reduction.
(4) A comprehensive evaluation of PHub in terms of performance, scalability, and deployment cost (§4).

2 Bottlenecks in Cloud-Based Training

Modern neural networks 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* [48], 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 gradient descent optimization algorithm. Computation is often done on GPUs or other accelerators suited to regular data-parallel operations, processing tens to hundreds of samples at once (*minibatching*).



Wi: Worker i, M: Model Size, N: Worker Count, T: Compute Time, Per machine bidirectional bandwidth (last row)

Figure 4: PS configurations in a DDNN training system, and minimum network bandwidth to fully hide communication overhead.

The distributed training process (Figure 3) is different in a few ways. First, a mean gradient is calculated across all minibatches in all the GPUs 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. Gradient aggregation and optimization are element-wise operations. Aggregation sums gradients from all workers. Optimization updates the model using aggregated gradients with an algorithm such as SGD. Our design goal is to overlap aggregation and optimization of different keys with communication. This paper focuses on optimizing calculation of both the mean gradient across machines and the subsequent model update (or *parameter exchange*).

In a typical DDNN training setup, machines can take the role of a worker and/or a parameter server (PS). PSs are specialized key-values stores that collect the gradients and update the model [40, 41, 53, 63]. In this paper, we use "key" to refer to a layer, and "value" to refer to the set of parameters for that layer.

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 [19, 21, 24, 26, 28, 46] or with relaxed consistency [25, 27, 32, 59], 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.

2.1 Common PS Configurations

PS configurations 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 4 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 [7, 13], 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 DDNN training.

2.2 The MXNet Framework

MXNet [22] is a state-of-the-art DDNN training framework that supports many new optimizations in the literature [23, 26, 64]. It is widely used on AWS [2], and natively supports distributed training: its PS implementation relies on TCP, and is built on top of the ZMQ [18] distributed messaging library.

All modern DNN training frameworks can fully utilize GPU resources by taking advantage of primitives, such as CuDNN. These frameworks offer comparable throughput when training DNNs. For distributed training, many frameworks such as MXNet provide eager scheduling of parameter exchanges, overlapping backward computation with parameter synchronization, hiding communication latency. We measured distributed training performance and scalability for Caffe2, TensorFlow, and MXNet[2] with up to 8 worker nodes. We found comparable throughput when training ResNet 50 on a 56 Gbps network using SGD, with MXNet slightly leading the pack (Table 1). These results align well with other observations [15, 51, 65]. Therefore, we use MXNet as the basis for our implementations and optimizations, but the techniques are generalizable.

2.3 Bottleneck Findings

Ideally, communication latency is fully hidden by computation, i.e., compute engines never wait for data. In reality, since computation speed exceeds communication speed in cloud-based DDNN training, time is wasted waiting for model updates (Figure 2). Workers run much faster locally (Table 1), so the bottlenecks must lie in the PS, the network stack, and/or the physical network itself. Our study finds three major bottlenecks in cloud-based DDNN training: insufficient network bandwidth, framework inefficiencies, and suboptimal deployment in the cluster. We elaborate on each below.

	Local	2 nodes	4 nodes	8 nodes
TensorFlow	152	213	410	634
Caffe2	195	266	343	513
TF+Poseidon[64]	209	229	364	<648
MXNet	190	187	375	**688**

Table 1: Throughput (samples/s) of training ResNet 50 on major DNN training frameworks with a 56 Gbps network.

2.3.1 Insufficient Bandwidth 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 4 gives an analytical lower bound of *per host bandwidth*, and Table 2 shows the required bandwidth for various DNNs: DNNs demand more bandwidth than mainstream cloud providers offer (typically 10-25 Gbps) in the VMs. A similar observation was made in prior work [20, 65]. Furthermore, bandwidth requirements increase with worker count.

2.3.2 Framework Bottlenecks However, even with ample communication resources, existing PSs failed to hide communication latency and struggled to scale. Table 1 shows that all major DNN training frameworks do not scale well with a 56 Gbps IPoIB network. We investigated the cause for MXNet by breaking down the overhead for each major component of a training iteration (legends of Figure 5). 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 5: we gradually turned on different components in the MXNet DDNN 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:

[2]Caffe2: Gloo Halving and doubling enabled. TensorFlow and MXNet: CS PSs with a 1:1 worker-to-PS ratio. Network: 56 Gbps IPoIB. GPU: GTX 1080 Ti. Neural Network: ResNet 50 with batch size of 32. Poseidon hangs when more than 5 workers are training this network in our cluster. 8-worker throughput is overestimated as per worker throughput (at 5 workers) * 8.

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: Estimated bisection bandwidth (Gbps) lower bound on the PS side for hiding communication latency. Same setup as Table 1.

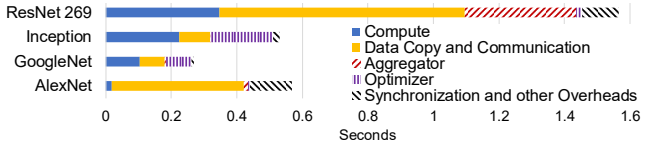


Figure 5: Progressive overhead breakdown of different stages during the distributed training pipeline for MXNet DDNN training on a 56Gbps network. Link capacity accounts for a small fraction of the copy and communication overhead in this setting.

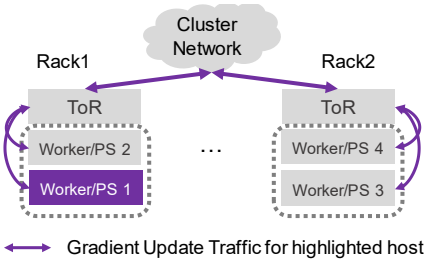


Figure 6: When workers/servers span multiple racks in cloud-based training, large delay occurs due to gradient update traffic going through an oversubscribed network core.

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 (see §3.2.2).
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.

2.3.3 Deployment-related Overhead VMs associated with a training job can be far away from each other when launched in the cloud. Existing frameworks assume homogeneous inter-VM bandwidths, which results in excessive communications between distant VMs, in turn leading to bottlenecks. We conducted an experiment to probe pair-wise bandwidth of 8 P2.8xLarge 10 Gbps instances on EC2 and found that bandwidths can vary by a factor of 2—even between send and receive streams of the same instance pair. Some cloud environments support co-scheduling constraints (e.g., EC2 placement groups), but for large jobs on busy clusters the scheduler may take a long time to satisfy these constraints.

One possible reason is a congested or oversubscribed network topology in the data center [43, 44, 47], providing full bisection bandwidth within each rack but not across racks when the core is busy. Thus, gradient update streams that go through a potentially oversubscribed network core [30, 52] can be delayed. Network topology awareness is crucial for DDNN workloads [54, 58]. In our work, we pursue a rack-scale PS that takes advantage of intra-rack full bisection bandwidth and minimizes inter-rack traffic via *hierarchical reduction algorithms* (see Section 3.4).

3 PHub Design

Based on §2.3 findings, we propose a rack-scale PS, PHub, that reduces framework overhead with software optimizations, mitigates bandwidth insufficiency with a re-architected, balanced server configuration, and lowers network environment-induced overhead with topology-aware reduction algorithms. With PHub, we aim to:

(1) Minimize gradient/model communication overhead.
(2) Enable efficient gradient processing and overlap with communication.
(3) Balance communication and computation capabilities, both within and PS and between workers and the PS.
(4) Allow low interference of co-running jobs and minimized cross-rack traffic.

3.1 The PHub Service API and Interoperability with other Frameworks

PHub's API is designed for compatibility with multiple DNN training frameworks. Workers use PHub 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. PHub 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::connect FullMesh` in Caffe2 and `GrpcServer::Init` in TensorFlow. `PHub::InitService` causes the current PHub instance to allocate and register receive and merge buffers. PHub also authenticates each worker's identity using the nonce. Authentication is a one-time overhead and once a connection is established, PHub assumes the remote identity associated with that address/port/queue number does not change during training.

PHub'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`). PHub 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`.

3.2 PHub Software Optimizations

This section describes software optimizations that benefit different stages in DDNN 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 [4]. Note that similar results could be achieved over Ethernet using RoCE, DPDK or other frameworks. We followed the guidelines from [36]; 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, PHub 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, PHub encodes metadata (such as callback ID and message opcode) into InfiniBand's queue pair number and immediate field. This saves PHub 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 [26]. 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 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 7 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

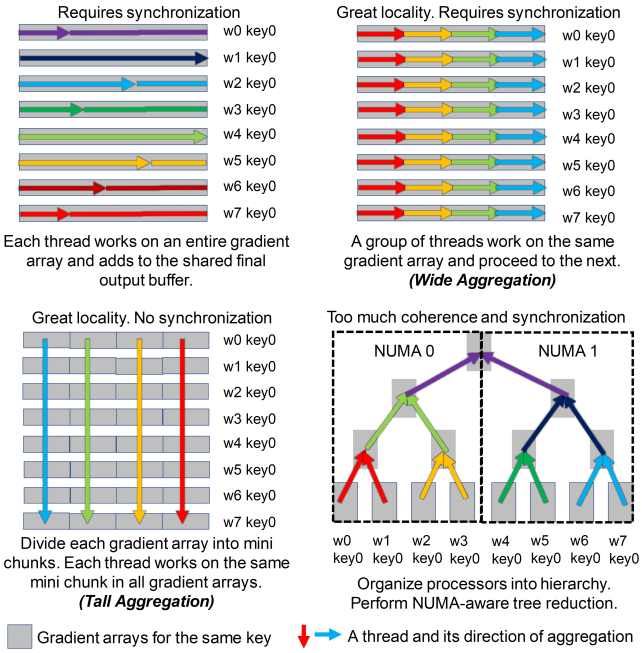


Figure 7: Ways of gradient aggregation. A thread (arrow) aggregates over the array (gray rectangle) of gradients from a worker.

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.

PHub 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 PHub: the thread that aggregates a particular chunk also optimizes that chunk. As a result, PHub's aggregation and optimization scheme effectively maps a particular chunk to a single core (since PHub pins threads to cores). On the other hand, MXNet

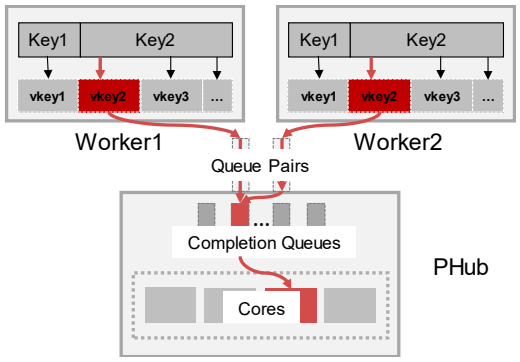


Figure 8: The process of mapping a chunk to a core in PHub using fine grained key chunking. Keys are chunked into virtual keys. The highlighted key is delivered to a highlighted (fixed) core through a highlighted (fixed) queue pair and completion queue.

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. PHub's aggregators and optimizers are fully extensible: implementations that comply with PHub's API can be used during runtime.

3.2.3 Fine-grained Key Chunking Chunking in PHub 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, PHub'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, PHub enables "streaming" aggregation and optimization.

3.2.4 Mapping a Chunk to a Core PHub'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 PHub assembles and disassembles a key, a process that is transparent to the framework.

PHub'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. PHub 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. PHub's chunk mapping mechanism is summarized in Figure 8.

3.3 A Balanced Hardware Design for Rack-Scale PSs

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 [35], despite the optimizations in §3.2. 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 bandwidth usage (Table 2) when serving multiple workers, starving the compute units.

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.

This section describes PBox, 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 prototyped PBox 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, PBox needed a matching network capability, which we provided by using multiple network interfaces. Figure 9 shows the resulting PBox 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 and provides roughly 136 GB/s bandwidth once IB and PCIe framing overheads are taken into account, balancing IO and memory bandwidth.

Hardware alone solves only part of the problem. Existing frameworks cannot efficiently use the full hardware capability even if multiple interfaces are present (for example, TensorFlow and MXNet support multiple interfaces only by

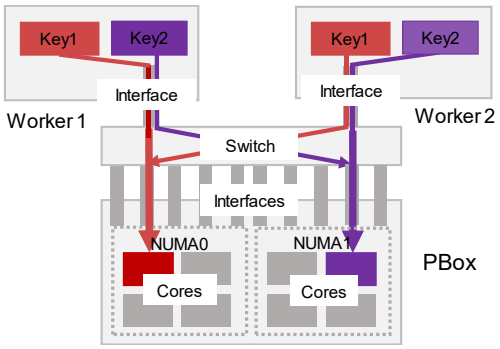


Figure 9: The PBox architecture

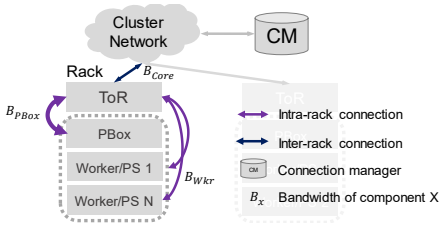


Figure 10: PBox deployment scheme

spawning multiple PS processes). Thus, software that understands both the hardware topology *and* balance is required to complete the solution. PHub takes full advantage of PBox by extending the chunk-to-core mapping scheme (§3.2.4), ensuring balance across interfaces and NUMA domains. PHub further guarantees no inter-processor traffic on PBox, 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, PBox forms micro-shards inside a box.

3.4 Rack Deployment and Topology-Aware Reduction

We associate a PBox 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 PBox, while oversubscription occurs between the ToR and the cluster network. Second, as we show in §4.7, a single PBox has enough scalability for a typical rack of worker machines.

When provisioned in each rack (Figure 10), PBoxes 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 PBox centrally aggregates gradient updates from workers in the same rack; then, the PBox nodes start cross-rack aggregation and compute globally aggregated gradients; finally, each per-rack PBox 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). PHub determines 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 PBox, 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). §4.8 estimates the overhead of C, and B_{Core} can be effectively probed by using [33, 34].

4 Evaluation

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

4.1 Experimental Setup

We evaluated our system with 8 worker nodes and one specially configured PBox 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 PBox 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. PBox 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 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.

4.2 DNNs Used in the Evaluation

We evaluated PHub's performance by training state-of-the-art deep neural networks using reference code provided with MXNet. We implemented a cache-enabled optimizer using

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

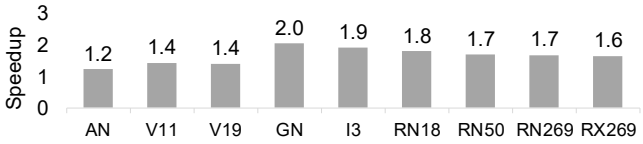


Figure 11: Speedup from a faster data plane that supports zero copy.

SGD with Nesterov's accelerated gradient method [45] and a cache-enabled aggregator for PHub. 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 [23] because they slow down training.

Table 3 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 change accuracy: they did not change the computations that were performed. We trained multiple DNNs to convergence to verify this.

4.3 Training Performance Evaluation

We include multiple results to highlight the effects of different software and hardware optimizations on PHub'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.

4.3.1 Benefit of a Faster Data Plane Figure 11 shows the performance of replacing the communication stack of the MXNet PS with a native InfiniBand implementation (MXNet IB) that had all optimizations noted in §3.2.1. This lets us see the benefit of switching to an optimized network stack without changing the PS architecture. We used our *enhanced baseline MXNet IB* in all the following evaluation.

4.3.2 Other Software and Hardware Optimizations We now quantify further benefits from PHub's software and hardware optimizations. We used CS MXNet IB in this comparison. PShard results were obtained by running PHub software on each worker as CS PSs. PBox results represent running PHub software on our single PBox machine as a NCC PS. We omit results for NCS and CC PSs for clarity. They performed similarly to PBox results.

Figure 12 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 PBox significantly affects runtime. In addition, we provide speedup when training with only 7 workers and PBox, *so that the total machine count in the system is equal to the baseline*.

Figure 13 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 PBox versus MXNet IB. Since PHub 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 IB signifies the benefit of software optimizations in §3.2.2-§3.3. The gap between PShard and PBox 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 optimizations in §3.3.

Figure 14 breaks down the overhead in different distributed training stages when running PHub in the same setup as Figure 5. Compared to Figure 5, *PHub reduces overheads from data copy, aggregation, optimization, and synchronization, and fully overlaps these stages, shifting the training back to a compute-bound workload.*

4.4 Performance with Infinitely Fast Compute

We used a benchmark to assess the efficiency of PHub'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 PHub.

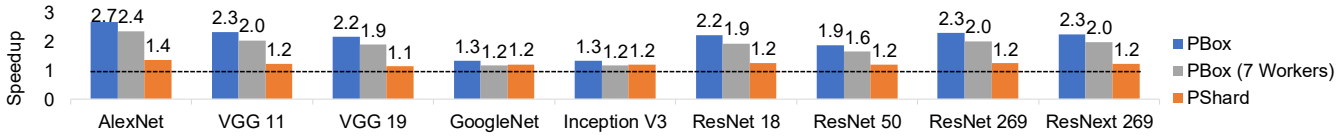


Figure 12: Training performance on a cloud-like 10 Gbps network. Results are normalized to sharded MXNet IB (*enhanced baseline*).

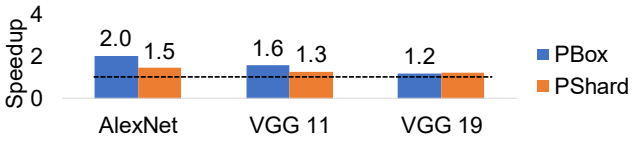


Figure 13: Training performance on a 56 Gbps network compared to MXNet IB (*enhanced baseline*). Computation speed bottlenecked training throughput for all but AlexNet and VGG.

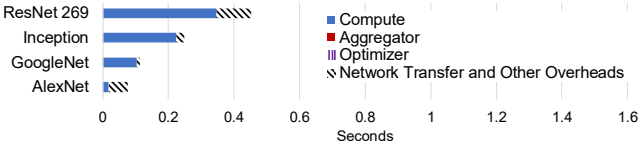


Figure 14: Progressive overhead breakdown of PHub. Compared to Figure 5, GPU compute time now dominates training time. Aggregator and optimizer have minimum overhead, and are barely visible.

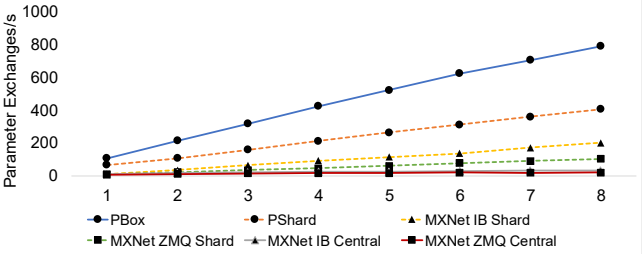


Figure 15: PBox provides linear scaling of throughput for 8 worker nodes with infinitely fast compute, training ResNet 18.

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

Figure 15 shows the results of running the benchmark with PBox, PShard and multiple baseline configurations. PBox provided linear scaling with 8 workers and outperformed the baseline by a large margin (up to 40x). PBox 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.

4.5 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 PHub's key assignment scheme and tall aggregation/optimization in leveraging locality.

Key Affinity in PBox: We evaluate two schemes for connecting workers to PBox to exploit locality and load balancing.

	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 4: Bidirectional memory bandwidth (GB/s) utilization in PHub 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.

In *Key by Interface/Core mode*, workers partition their keys equally across different interfaces on the PBox. 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 PHub exploit locality within a single worker. It also provides naturally perfect load balancing across interfaces and cores at the cost of additional communication 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 PHub: Caching benefits many PHub 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 PHub by measuring memory bandwidth usage.

Table 4 shows the memory bandwidth costs of communication, aggregation, and optimization on PBox. 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.

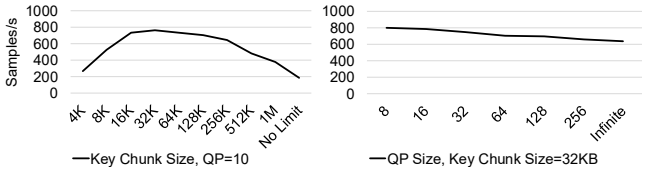


Figure 16: Effect of chunk size and queue pair count on throughput.

Without aggregation and optimization, PBox's bidirectional memory bandwidth usage was stable at 77.5 GB/s. No cache was used in this case because PBox 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 PHub 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.

4.6 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: PHub leverages fine-grained key chunking to better balance load and overlap gradient reception and aggregation. Figure 16 (left) evaluates the effect of chunk size with `ZeroComputeEngine` on PBox. 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 PBox 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 16 (right) evaluates the tradeoff, showing that fewer queue pairs was optimal.

4.7 Limits on Scalability

The scalability of PHub is inherently limited by available total memory, network or PCIe bandwidth. This section explores how close PHub gets to these limits. We use PBox to answer these questions. PBox 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

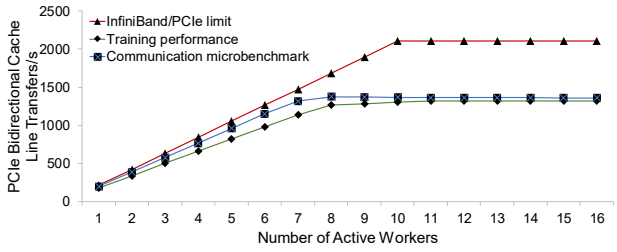


Figure 17: PBox scalability is limited by the throughput of the PCIe to the on-chip network bridge of the PBox processors. PHub can utilize 97% of the measured peak PCIe bandwidth.

connections to the `ib_write_bw` process on PBox. 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 PHub system is the PCIe-to-memory-system bridge.*

Figure 17 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 PHub can run during training. We also see that, when training VGG with `ZeroComputeEngine`, as more workers are added, PBox's performance approached the microbenchmarks (97%), demonstrating PHub's ability to fully utilize system resources. The gap in the plot between PBox and the microbenchmark is due to the overhead of scheduling operations in MXNet and straggler effects in workers. PBox hit the limit at a sustained 80GB/s memory bandwidth.

In real training, however, PBox's scalability limit was difficult to reach. Recent work ([37, 39]) 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 [38, 62]. One example [29] 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 PBox could reach this scale, we measured the memory bandwidth usage for ResNet 50 with 8 workers using the same batch size. We found that PBox required only

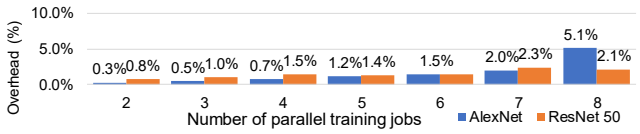


Figure 18: Overhead of multiple parallel training jobs sharing the same PBox instance.

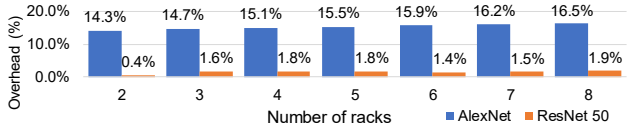


Figure 19: Emulated overhead of hierarchical reduction with PBox.

6GB/s memory bandwidth and an aggregated 4GB/s network bandwidth. This suggests that our PBox prototype 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 [1] processors provide nearly triple the Stream Triad performance (290 GB/s) [9] and 40% more PCIe bandwidth than our PBox machine. We would expect Epyc to support 40% more throughput.

4.8 Effectiveness of PBox as a Rack-Scale Service

We now evaluate effectiveness of PBox 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 PBox instance with different key namespaces and (2) when a training job crosses rack boundaries, and PHub performs hierarchical reduction.

Figure 18 shows the overhead of running multiple independent training jobs sharing a single PBox. 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 19 emulates a single cloud-based training job whose VMs span N racks, and each rack contains 8 workers and 1 PBox. The PBox uses a widely used ring reduction algorithm [5, 50] for inter-rack aggregation.

Since we have only one PBox 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 PHub'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.

4.9 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, based on the data in Figure 1. 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 [17] without GPUs; the GPU price G is (\$699 [14]). The 100Gb baseline uses Mellanox ConnectX-4 EN cards (\$795 [12]) and 2m cables (\$94 [11]). The 25Gb PHub workers use Mellanox ConnectX-4 Lx EN cards (\$260 [12]) and 4-to-1 breakout cables (\$31.25 per port [11]). The PHub node (also same as evaluation) cost H is \$8407 [16], plus 10 dual-port 25Gb Mellanox ConnectX-4

	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 5: 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.

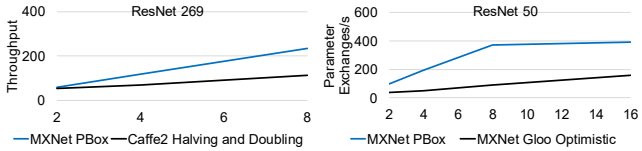


Figure 20: Left: Comparing Caffe2 + Gloo and MXNet + PBox on an 10Gbps InfiniBand network. Right: Comparing MXNet + Gloo and MXNet + PBox on a 56Gbps InfiniBand network with `ZeroComputeEngine`.

Lx EN cards ($162.5 per port [12]). 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 PHub ratio).

We use the Arista 7060CX-32S 32-port 100Gb Ethernet switch ($21077 [3]) 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 PHub and 65 25Gb workers; with 3:1, 76.

Table 5 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.

5 Related Work

Other Communication 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 [35, 57].

To understand how PHub compares to other communication schemes, we first ran Caffe2 and MXNet with PBox. We used InfiniBand for both systems. We evaluated the fastest algorithm in Gloo: recursive halving and doubling, used in [29]. Figure 20 (left) shows PBox was nearly 2x faster.

We ported Gloo to MXNet to better assess both systems. Gloo implements blocking 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 20 (right) shows PBox 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` [57]). Second, collectives frequently use multi-round communication schemes whereas PBox uses only 1 round.

Compression, Quantization, Sparse Vector Communication, and Other Mechanism for Traffic Reduction: Orthogonal to our work are techniques to reduce gradient traffic. These techniques trade higher overhead in preparing and processing network data for lower network bandwidth usage. For example, MXNet supports a 2-bit compression scheme, similar to [49]. We compared PHub running on PBox to MXNet IB with 2-bit compression: PBox without compression still beat MXNet IB by 2x.

Other examples include Sufficient Factor Broadcast (SFB) [60, 64], which decomposes the gradient of a fully connected layer (FCL) into the outer product of two vectors. SFB uses a P2P broadcast scheme whose overhead scales quadratically with the number of machines, making it suboptimal for large scale training. Project Adam [24] sends activation and error gradient vectors for reconstruction on server. Both techniques have limited applicability as they only apply to FCLs, which are small or unused in recent neural networks [31, 55, 56, 61].

PHub can also work with gradient compression [42] to gain further benefits from its low latency communication stack, fast aggregation and optimization.

6 Conclusion

We found that inefficient PS software architecture and network environment-induced overhead were the major bottlenecks of distributed training with modern GPUs in the cloud, making DDNN training a communication-bound workload. To eliminate these bottlenecks, we proposed PHub, a high performance multi-tenant, rack-scale PS design, with co-designed software and hardware to accelerate rack-level and hierarchical cross-rack parameter exchange. Our evaluation showed that PHub provides up to 2.7x higher throughput, with 25% better throughput per dollar.

Acknowledgments

We would like to thank members of Sampa, SAML and Systems groups at the Allen School for their feedback on the work and manuscript. This work was supported in part by NSF under grants #1723352 and #1518703 and gifts from Intel, Oracle and Huawei.

References

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

[2] Apache mxnet on aws. https://aws.amazon.com/mxnet/. (Accessed on 05/09/2018).

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

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

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

[6] Cloud tpus - ml accelerators for tensorflow Âăl Âă google cloud. https://cloud.google.com/tpu/. (Accessed on 05/16/2018).

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

[8] Ec2instances.info Easy Amazon EC2 instance comparison. https://www.ec2instances.info/?region=us-west-2.

[9] Epyc benchmarks. https://www.amd.com/en/products/epyc-benchmarks.

[10] Machine learning | microsoft azure. https://azure.microsoft.com/en-us/services/machine-learning-studio/. (Accessed on 05/16/2018).

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

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

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

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

[15] Performance of distributed deep learning using ChainerMN. https://chainer.org/general/2017/02/08/Performance-of-Distributed-Deep-Learning-Using-ChainerMN.html.

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

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

[18] ZMQ distributed messaging. http://zeromq.org/.

[19] 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.

[20] Michael Alan, Aurojit Panda, Domenic Bottini, Lisa Jian, Pranay Kumar, and Scott Shenker. Network evolution for dnns.

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

[22] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, Dave Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: A scalable and flexible data center network. Association for Computing Machinery, Inc., August 2009.

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

[32] 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.

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

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

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

[40] 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.

[41] 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.

[42] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J. Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. *CoRR*, abs/1712.01887, 2017.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Neurocomputing: Foundations of research. chapter Learning Representations by Back-propagating Errors, pages 696–699. MIT Press, Cambridge, MA, USA, 1988.

[49] 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.

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

[51] Shaohuai Shi, Qiang Wang, Pengfei Xu, and Xiaowen Chu. Benchmarking state-of-the-art deep learning software tools, 2016.

[52] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs HÃűlzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of clos topologies and centralized control in googleâĂŹs datacenter network. In *Sigcomm '15*, 2015.

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

[54] Christopher Streiffer, Huan Chen, Theophilus Benson, and Asim Kadav. Deepconfig: Automating data center network topologies management with machine learning. *CoRR*, abs/1712.03890, 2017.

[55] Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. Inception-v4, inception-resnet and the impact of residual connections on learning. *CoRR*, abs/1602.07261, 2016.

[56] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567, 2015.

[57] 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.

[58] Leyuan Wang, Mu Li, Edo Liberty, and Alex J Smola. Optimal message scheduling for aggregation. *NETWORKS*, 2(3):2–3, 2018.

[59] 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.

[60] Pengtao Xie, Jin Kyu Kim, and Eric P. Xing. Large scale distributed multiclass logistic regression. *CoRR*, abs/1409.5705, 2014.

[61] Saining Xie, Ross B. Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. *CoRR*, abs/1611.05431, 2016.

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

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

[64] 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.

[65] H. Zhu, M. Akrout, B. Zheng, A. Pelegris, A. Phanishayee, B. Schroeder, and G. Pekhimenko. TBD: Benchmarking and Analyzing Deep Neural Network Training. *ArXiv e-prints*, March 2018.