PLink: Efficient Cloud-based Training with Topology-aware Dynamic Hierarchical Aggregation
Liang Luo, Peter West, Jacob Nelson, Arvind Krishnamurthy, Luis Ceze

Abstract
Training deep learning (DL) models has become an important workload on the commercial cloud. Scaling cloud-based distributed training faces unique challenges from the hierarchical network topology of the datacenter and the dynamic nature of the multi-tenant environment. Timely training of deep learning models requires effective use of topology-induced locality in the datacenter network.

This work proposes PLink, an optimized communication library that probes the physical network and then generates and executes a fitted hierarchical aggregation plan to take advantage of locality in the datacenter network. Further, it adapts to changing network conditions by dynamically selecting aggregation routes. PLink needs no support from cloud providers and operates out-of-the-box on unmodified commercial clouds. PLink techniques deliver up to 95% accuracy inferring VM physical affinity. Crucially, PLink serves as a direct plug-in to many training frameworks, delivering up to 3x faster end-to-end training performance for popular DL models on Azure and EC2 compared to the state of the art.

1 Introduction
Speedy training of complex models requires exploiting distributed training. The distributed training pipeline consists of two stages: local computation (as part of forward and backward passes) and global communication (gradient aggregation and optimization). Scaling up distributed training involves making both stages efficient. Past work in distributed DL training has focused on building specialized hardware clusters with quick interconnects to tackle these challenges [22, 42, 49, 52, 67, 81, 84, 89]. While the results have been encouraging, these approaches demand steep investments and are not available to everyone.

On the other hand, cloud-based learning has become a popular, more accessible alternative. Today, all major commercial cloud providers offer racks of nodes with specialized accelerators (such as GPUs, TPUs, and custom FPGAs) [3, 6, 10, 12, 40, 53, 87] for deep learning workloads.

Existing solutions assume uniform link bandwidth and exclusive use of the network infrastructure. These assumptions, however, do not hold for datacenter networks used in commercial clouds. The hierarchical network structure, multi-tenancy and the dynamic nature of the cloud traffic workload, the unpredictability of the locations where VMs are spawned, and common design choices such as oversubscription in the datacenter network, all add to the complexity of scaling up cloud-based distributed training and reduce the effectiveness of existing communication solutions [62, 63].

Our work focuses on designing a communication scheme for efficient gradient aggregation in the context of datacenter networks that appear in commercial clouds. We address three specific challenges in developing our communication scheme. First, we need an aggregation mechanism that is appropriate for network topologies that display bandwidth oversubscription, and that makes appropriate use of under-provisioned links. Second, we need to be able to identify the underlying network topologies, locality characteristics, and bandwidth constraints even if the commercial cloud does not expose such information due to security and business considerations. Finally, we need communication schemes that can react to changing network conditions, especially in the presence of interfering traffic generated by other tenants on the commercial cloud.

We propose PLink, an optimized, topology-aware, dynamic system that uses a hierarchical aggregation scheme for cloud-based, high performance DL training. PLink uses three components to address the challenges listed above: (1) ProbeEmbed (§ 4.1), a general purpose technique that uses data from measurement probes to embed cloud-assigned VMs in a multi-dimensional space in order to accurately infer and group VM nodes based on their physical affinity; (2) AggEngine (§ 4.2), a hierarchical aggregation execution engine codesigned with the DL training workload that generates and executes a balanced hierarchical aggregation plan using topology information; (3) Autotune (§ 4.3), a reactive mechanism to quickly adjust to network changes by rebalancing the aggregation workload between VMs and continuously fine-tuning aggregation communication paths to match each VM’s current network performance.

PLink serves as a direct plug-in to popular training frameworks and requires no support from cloud providers. On unmodified commercial clouds (Azure and EC2), PLink is able to achieve up to 95% accuracy in inferring physical affinity, up to 3x faster aggregation performance, and up to 3x faster end-to-end training performance for popular DL models compared to the state of the art.

2 Background
This section contextualizes distributed training in the cloud environment. We provide a quick overview of typical structures for datacenter networks and common approaches to the communication phase of training.

2.1 Datacenter Networks
A typical datacenter network has a hierarchical, multi-tiered topology [44, 61, 69, 74]. Tens of machines are grouped into
We now describe distributed training, particularly the pre-
with their locally-produced gradients. Most modern training frame-
ward pass and a backward pass to derive gradients for model
aggregated (summed in order to compute the average).

Non-uniform link bandwidth. Host-to-host bandwidth in
hosts is largely affected by where they reside: for example,
two end-hosts in the same rack are guaranteed full link bi-
section bandwidth, because link capacity is not shared at the
rack-level; whereas the communication performance between
two hosts on different racks depends on two important fac-
tors: link congestion, and the oversubscription ratio. Although
some cloud providers have mechanisms for enforcing locality
(e.g., placement groups on EC2), they usually impose capacity
limits [19] and such mechanisms are not universally available.
In this paper, we use the term topology-induced locality to refer
to the variation in communication performance deter-
mined by where the VM nodes physically reside. Efficient
communication in the cloud requires carefully architecting
software to tap into locality [1, 51], avoiding large transfers
on high latency bottleneck links.

2.2 Distributed DNN training

We now describe distributed training, particularly the prev-
vailing paradigm of synchronous data parallelism. In this
paradigm, each worker processes a portion of the dataset.
Training is done in iterations, and each iteration processes a
batch of data. An iteration consists of two steps: the first
step is a computationally expensive phase involving a for-
ward pass and a backward pass to derive gradients for model
update; the second step is the communication-intensive par-
parameter exchange phase, where gradients from all workers
are aggregated (summed in order to compute the average).
The averaged gradients are then used to optimize the current
model, after which the next iteration starts. Iterations on all
workers happen in lock step. Most modern training frame-
works can hide a portion of the latency of parameter exchange
by overlapping it with gradient computation. Broadly, there
are three ways to exchange parameters (but this is not a strict
taxonomy as some approaches can be in more than one cate-
gory):

Parameter Servers (PS) [37, 57, 58, 63, 82, 90, 91]. PSs are
key-value stores, where keys and values represent the model’s
layer IDs and weights. PSs can be centralized or sharded. In
each iteration, all workers update the model stored in PSs
with their locally-produced gradients.

Collective AllReduce (CA) [25, 28, 72, 75, 85]. Popular in
the context of MPI, all nodes in CA participate in the com-
munication, usually running symmetric tasks. The end goal of
CA is that all nodes have a globally-reduced copy of the data.

Hierarchical Aggregation (HA). Pervasive in the HPC
world [43], HA 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 [35, 41, 49],
though not in a datacenter context.

The existing approach to mapping a given communication
pattern to the cloud infrastructure is straightforward. The user
requests a set of VM nodes from the cloud provider directly
or indirectly from a service (e.g., Batch AI [4], Dynamic
Training [11]); then the list of node addresses are used to
launch a deep learning framework [14, 20, 21]. Therefore, this
randomly-ordered list determines the identity of each node,
which dictates the communication pattern.

3 Motivation

This section summarizes major challenges to scale up cloud-
based DL training, and motivates the use of 2LHA, a form of
HA to speedup training.

3.1 Challenges

We describe a few major challenges in scaling cloud-based
distributed training:

Non-uniform link bandwidth. Host-to-host bandwidth in
the cloud is non-uniform due to the hierarchical structure of
the datacenter (§2.1), as VM nodes can be spread across
different physical hosts, racks, rows or even clusters. Figure
1-left shows a pair-wise bandwidth probe of 32 C5.9xlarge
nodes launched in EC2’s US-east-1 region, in the same avail-
ability zone, and Figure 1-right shows F16 instances launched
in Azure’s US-west-2 region. In both cases, we observe faster
pairs can deliver more than 2x the bandwidth of slower pairs
during iPerf bandwidth probes.

Volatile traffic. The performance variability in the commer-
cial cloud, in terms of bandwidth and latency variance, is well
known [39, 50]. Although multiple previous works [79, 80]
have proposed mechanisms for performance isolation, emp-
pirically we observe interference from other workloads. The
available bandwidth between the same pair of nodes can vary
with time, as a consequence of changing workloads in the
datacenter, noisy colocated VMs, or migration.

3.2 Inefficiencies in Existing Approaches

We motivate our design by demonstrating why some existing
approaches do not perform optimally, as they rely on assump-
tions that aren’t typically valid in the datacenter setting.

Figure 2 shows a theoretical analysis of widely used com-
unication 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 schedules that achieve optimal locality in the setting by
exchanging data among nodes with high locality (green high
performance links), while minimizing transfers over the bottleneck links (red slow links). The right side shows alternative reduction routes with poor locality. Note that all schedules achieve the same result, albeit at different efficiency levels.

The problem with poor locality arises as soon as the communication pattern in the algorithm is not optimally aligned with physical topology. For example, in the case of the bad recursive halving-doubling schedule highlighted in Figure 2(b-right), it would be much better if the IDs of node 1 and 2 are swapped, creating the schedule on the left. Being able to do so is contingent on having a good understanding of the physical network. 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 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 [41, 49, 63]. 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 latency is unpredictable. Tree reduction inherits the problems of both PS and CA: when the fan-out is large, the upward aggregation node can experience incast congestion, but when fan out is small, the aggregation takes more rounds to finish. To meet the characteristics of the datacenter network, we need an algorithm that bounds communication steps, takes advantages of fast links, and localizes traffic to avoid interference from competing traffic.

<table>
<thead>
<tr>
<th>Name</th>
<th>Rounds</th>
<th>Bytes</th>
<th>XR Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS (fully sharded)</td>
<td>2</td>
<td>2S(NC – 1)</td>
<td>2NS(C – 1)</td>
</tr>
<tr>
<td>Halving doubling</td>
<td>2log₂NC</td>
<td>2NCS</td>
<td>2CS</td>
</tr>
<tr>
<td>Ring-Chunked</td>
<td>4NC</td>
<td>2NCS</td>
<td>CS</td>
</tr>
<tr>
<td>Tree (fan out = C)</td>
<td>2log₂NC</td>
<td>2NSC/NC / C</td>
<td>≈ 2 NC/NC</td>
</tr>
<tr>
<td>2-level hierarchical</td>
<td>4</td>
<td>2S(NC – 1)</td>
<td>2S(C – 1)</td>
</tr>
</tbody>
</table>

Table 1: Network characteristics of various algorithms featuring rounds of communication (Rounds), minimum total traffic (Min Bytes), and minimum cross rack traffic (Min XR Bytes) to allreduce with NC nodes on C racks, each with N nodes. Each node has a buffer of size 5 bytes. PS and aggregators in HA are colocated and sharded.

### 3.3 2-level Hierarchical Aggregation

Compared to CAs, HA has the potential to fit nicely with the physical topology of a datacenter network, provided that the network topology is known. HA does not reduce the total amount of data transferred on wire, but instead has the ability to achieve more *localized* traffic and considerably smaller transfers on slow links.

We use a form of HA, a 2-level hierarchical (2LHA) aggregation scheme, to adapt to the datacenter network topology. We are particularly interested in 2-level aggregation because it strikes a balance between smaller latency (requiring fewer rounds) and more aggressive traffic localization (requiring more rounds).

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

---

**Figure 1**: Pairwise bandwidth probes with 32 EC2 C5.9xlarge and Azure F16 instances show non-uniform link bandwidth.

**Figure 2**: Existing aggregation approaches suffer from poor locality if the communication pattern of the algorithm does not align well with the underlying physical network topology. Some steps omitted for clarity.
We now describe PLink, an optimized, topology-aware, and three components to achieve this: address the major challenges highlighted in 3.1. PLink uses training. To optimally utilize datacenter networks, PLink must dynamic system that leverages HA for efficient cloud-based

4 Design and Implementation

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. These optimizations are detailed in §4. Table 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.

4.1 Capturing Network Locality with ProbeEmbed

For accurate and efficient network topology discovery, ProbeEmbed must probe quickly and should not rely on knowledge of a particular datacenter. ProbeEmbed achieves this in three steps: (1) probing communication links between nodes to measure pairwise node distances, (2) denoising probed distances, and (3) clustering nodes.

4.1.1 Running ProbeEmbed probes

ProbeEmbed starts by issuing measurements to identify communication locality and determine pairwise node distances. ProbeEmbed defines distances using universal networking concepts, like latency or an inverse form of bandwidth. ProbeEmbed supports using a wide range of standard probing tools including ping, iPerf, NTTTCP [13], and Packet Trains [48]. We generally find latency-based probes better capture topology (§5.4). To provide near bare-metal latency measurements, we also implement a DPDK-based echo for latency probing on supported VMs in Azure and EC2. ProbeEmbed runs these networking probes in two paradigms.

One-to-one probes are multi-round tests that pairs of VM nodes participate in each round. \( O(N^2) \) time is required to probe \( N \) nodes if run sequentially. To accelerate this process, ProbeEmbed uses a scheduling algorithm to pick as many pairs as possible in each round, without having a node appear twice in each round to avoid interference from concurrent tests. This allows ProbeEmbed to probe in \( O(N) \) rounds.

All-to-all probes are one-shot tests that all nodes participate in. ProbeEmbed use these tests to assess distances of nodes in a more dynamic fashion under loaded conditions.

ProbeEmbed derives pairwise distances with probe results. In the case of bandwidth measurements, it converts bandwidth to distance by taking the inverse [2]. ProbeEmbed then proceeds to denoise the collected data, before clustering nodes into different groups based on their physical affinity.

4.1.2 Denoising probe data with embedding

ProbeEmbed aims to discover network topology, in terms of their physical affinity using probes in a potentially noisy environment. To do so, ProbeEmbed embeds nodes in a Euclidean coordinate space, obtaining a set of coordinates whose distances agree with the probed distances. This lets us:

1. Denoise measurements by leveraging Euclidean space, which we use to approximate the physical location of nodes inside a datacenter.

• AggEngine: a high performance implementation of 2LHA that is codesigned to take advantage of deep learning properties. AggEngine uses clustering information to efficiently distribute the aggregation workload and execute the aggregation schedule.

• Autotune: an adaptive mechanism that tracks training performance and adjusts the current GM and LM assignments to adapt to changes in the network conditions.
2. Obtain a set of “virtual coordinates” that will be used by a clustering algorithm to identify aggregation groups.

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

\[
\sum_{i=1}^{n} \sum_{j=1}^{n} ((d_{i,j})^\alpha + \mathbb{I}_h \|h_i + h_j\| - p_{i,j})^2
\]  

(1)

where \(n\) is the number of nodes, \(d_{i,j} = \|v_i - v_j\|\) is the Euclidean embedding distance between node coordinates for nodes \(i\) and \(j\), \(p_{i,j}\) is their probed distance, parameter \(\alpha\) takes a value between 1 and 2, \(h_i\) is a non-negative startup cost parameter for node \(i\), and \(\mathbb{I}_h\) is a switch for \(h_i\) (detailed below). We use the Adam algorithm \([54]\) 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 VM node are probably close to each other 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 in a set.

The parameter \(\alpha\) lets ProbeEmbed tune how longer distances are treated in the Euclidean space: \(x = 1\) fits the embedded distances to probe distances exactly. For \(\alpha > 1\) (e.g., \(\alpha = 2\)) we achieve two effects illustrated in Figure 4: increased compaction and focus for longer distances.

Probe distances might not directly reflect the physical distance we aim to capture. A small physical distance can be magnified disproportionately in the probe due to competing traffic. We mitigate this with parameter \(\alpha\), as shown in Figure 4 left. Setting \(\alpha = 1\) has the embedding directly capture probe values. Setting \(\alpha = 2\), these values are effectively “compacted”, with a much larger probe distance being represented by a moderately larger embedding distance (see \(p = 1\) vs. \(p = 5\) in Figure 4 left).

It is important to capture long distances as accurately as possible because they are usually the bottleneck in 2LHA. Setting \(\alpha > 1\) lets ProbeEmbed focus more on the representation of long distances in the embedding process. Figure 4 (right) shows that in the \(\alpha = 1\) case, the error for deviating from the optimal value of embedding distance \((d)\) is independent of probe value \(p\), but with \(\alpha > 1\), the same deviation for a longer probed distance results in a larger error in the optimization objective. Blowing up the error faster for fitting longer probed distances forces the optimizer to focus on them.

Inspired by \([38]\), ProbeEmbed includes an optional parameter \(h_i\), to represent the node-specific, network-agnostic, fixed latency of sending a packet. \(h_i\) models communication overhead that involves an upfront cost (e.g., a ping packet needs to traverse the operating system stack, which is not relevant to how far away VM nodes are). We enable \(h_i\) for ping measurements, and disable it otherwise.

![Figure 4: Effect of parameter \(\alpha\) on (left) optimal fit Euclidean distance \(d\) for probe \(p\) and (right) how optimization error blows up with deviation from optimal fit.](image)

ProbeEmbed incorporates multiple measurements, by selecting the lowest observed latency.

### 4.1.3 Grouping Nodes for 2LHA

We now outline how ProbeEmbed partitions VM nodes into groups for 2LHA. ProbeEmbed starts by determining the number of groups to generate, with the goal to generate groups that have balanced number of nodes to avoid stragglers.

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

\[
b = O\left(\frac{n}{k} + k\right)
\]  

(2)

where \(n\) is total nodes, and \(k\) is number of groups. The GM 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, then sends aggregated values back along those paths. This expression achieves a minimum at \(k = \sqrt{n}\), giving a simple motivation for choosing optimal cluster size.

When topology information is available through ProbeEmbed, to arrive at balanced, locality-preserving groups, we use the constrained k-means clustering algorithm with k-means++ initialization \([24, 30]\). This accepts a minimum cluster size as input, as well as the number of groups to generate. 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. Imposing perfect balance would mean that some group contains 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 (the largest group is no more than 2x the size of the smallest).

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 Efficient HA with AggEngine

AggEngine transforms grouping information from ProbeEmbed into a hierarchical reduction plan and efficiently executes it. AggEngine supports various communication backends including TCP, (Soft)RoCE [66], (Soft)iWarp [66], and InfiniBand, to maintain compatibility with all environments. We now describe AggEngine in the context of TCP.

4.2.1 Generating an Aggregation Plan.

AggEngine chunks oversized buffers into smaller, predefined sizes so load-balancing them across different processor cores is easier, as layers of a neural network can have vastly different sizes. Finer-grained chunks allow better overlapping of the transmission of gradients with aggregation, but at the cost of potentially more packets. We find chunk sizes between 32–64KB to be optimal.

AggEngine assigns chunk GMs and LMs in a balanced manner. For a given chunk, since more bandwidth is required for GMs, each group should have the number of GM nodes proportional to their cardinality, as 2LHA is bottlenecked by the slowest group. AggEngine uses an approximation set partition algorithm to achieve this. Assignment of LM placements in each group follows a similar fashion.

With GMs and LMs assigned, AggEngine generates a schedule that executes the steps in §3.3 for each chunk. A schedule consists of sequence of chunk-action pairs, where action is one of the following primitives1:

- **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 from *nids* have been received and aggregated into the merge buffer.

- **Fetch**: a fetch action is taken when the calling framework notifies AggEngine that a buffer can be read.

- **Deliver**: writes the content in the merge buffer back to the framework-supplied buffer.

4.2.2 Executing an Aggregation Schedule

When a job starts, AggEngine creates two merge buffers for each chunk (details below) and one receive buffer for each peer. Merge buffers are padded to the nearest cache-line size for efficient reduction using SSE or AVX. AggEngine then performs rendezvous with an out-of-band mechanism (a centralized Redis server). AggEngine establishes multiple connections per pair of VM as cloud providers can restrict per-stream bandwidth [19].

---

1With these primitives, AggEngine can support arbitrary reduction graphs.
late reduce call. AggEngine bookmarks the incoming chunk ID to a deferred queue, which is polled frequently to check if any progress can be made.

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

### 4.2.3 Changing an Aggregation Schedule

AggEngine needs to dynamically swap in new aggregation plans to react to network changes. One requirement for correctness in traditional key-value systems is maintaining coherence, which implies that all nodes must see the same model in each iteration. While a synchronous training process automatically guarantees coherence, maintaining it during a change of schedule requires heavy synchronization. Fortunately, maintaining coherence is optional in training, as much evidence [36, 46, 71, 73, 86, 88] shows it is not required to achieve high accuracy.

Schedule changes start by blocking subsequent calls to reduce. AggEngine attempts to bring its polling threads to a safe point, so that aggregation schedule for all buffers can be reset to Fetch stage. A safe point is reached by first draining the send queue so that no lingering buffers are being transferred, avoiding transferring old data when a new schedule is installed. Each buffer is given a 5s timeout to transition into the Fetch state gracefully. AggEngine assumes that the remote node did not send data if the timeout expires, and it proceeds to transition the buffer to Fetch state. AggEngine then signals to its peers that it is ready to switch to a new schedule for that chunk. AggEngine finally checks if all its peers are ready, clears the streaming cursor, and then installs a new schedule. AggEngine recovers chunks that are affected using chunkHolder, and requeues these buffers into the ready queue.

AggEngine thus supports a transparent procedure for switching to a new aggregation schedule, with the end result being that chunks whose reductions are interrupted by schedule changes may hold values from the previous iteration, while chunks whose reductions are uninterrupted reflect the new values of the current iteration.

Table 2 summarizes how AggEngine is designed to take advantage of properties in the distributed training workload to lower its overhead.

### 4.3 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 balance VM loads and compensate for link changes by redistributing LMs and GMs to nodes, so the time to finish an iteration is similar at both node and group levels.

A perfect initial LM and GM assignment is hard, even if we have bandwidth probe measurements. Consider an aggregation plan $p$, where the effective bandwidth of node $i$ to $j$ while running aggregation $p$ is $BW(i, j)$. Clearly, finding the best $p$ analytically relies on $BW(i, j)$ to be precisely measured or modeled, but $BW(i, j)$ has a circular dependency on $p$ itself, because it depends on the timing and concurrency of transfers. Meanwhile, $BW(i, j)$ might change due to competing traffic. To counteract this effect, the number of bytes transferred from $i$ to $j$ according to $p$, denoted by $D(i, j)$, should also reflect the change. Moreover, the time to compute model updates in different VM nodes may also differ, creating the straggler effect and adding to the complexity.

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 immediately adapt to the network change; and (2) a Finetune phase where Autotune uses a performance model to pinpoint the current performance bottleneck in the system, and move assigned GMs and LMs away from it in a step-by-step, increasingly aggressive manner.

#### 4.3.1 Quicktune

Quicktune is built on the fact that an efficient 2LHA schedule should be balanced at all levels. Quicktune aims to take a one-shot approach to globally balance GM and LM assignments according to the current effective bandwidth of each connection, while temporarily ignoring straggler effects coming from the compute pipeline. A formal problem definition of assigning GMs and LMs in the Quicktune realm is 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. We minimize the maximum per-node transfer time:

$$
\text{minimize} \quad \max(t_i = \sum_c S(c)(LM(i, c)G(i) + GM(i, c)G)) / \sum_n BW(i, n)$$

subject to:

$$\forall i, c \quad GM(i, c) = 1 \implies LM(i, c) = 1$$

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

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

While a conventional solver can be used, Quicktune accelerates this process by using an efficient approximation. Quicktune first distributes GMs to different groups, with the number of GMs assigned to each group proportional to the group bandwidth min-cut $\sum_{m \in G(i)} \sum_{n \in G(g)} BW(m, n)$. Finetune then distributes LMs inside each group to different members in a

<table>
<thead>
<tr>
<th>Property</th>
<th>AggEngine Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed comm. pattern</td>
<td>No explicit acknowledgement</td>
</tr>
<tr>
<td>Fixed buffer size</td>
<td>Minimal metadata</td>
</tr>
<tr>
<td>One reduction per layer per iteration</td>
<td>Only 2 merge buffers per layer; eagerly accepts chunks</td>
</tr>
<tr>
<td>Training is stochastic</td>
<td>Switching plans quickly and cheaply</td>
</tr>
</tbody>
</table>

Table 2: Codesigning AggEngine with training workload.
similar fashion, using aggregate per node bandwidth.

To retrieve $BW(i,j)$, Autotune probes the TCP_INFO [15,65] structure for delivery_rate to retrieve the per-link bandwidth directly from the OS. Since this bandwidth an average, Autotune filters by the app_limited field: bandwidth measured only when the outbound throughput is not throttled by the sending application is recorded. To further reduce averaging effects, Autotune queries this information after each successful write call.

4.3.2 Finetune

Quicktune’s theoretical analysis may not perfectly reflect the reality of the complex system; in particular, it ignores bandwidth and assumes bandwidth does not change under the new schedule. Finetune, however, approaches this by evolving the current schedule, using both the currently measured $D(i,j)$ and $B(i,j)$. Finetune pinpoints the current bottleneck node in the system, and then increasingly more aggressively moves away its load while maintaining balance. The core idea of Finetune is blame. Blame for node $B(i)$ has two parts: time $t(i)$ plus imbalance $b(i)$. For each node, Autotune collects essential information including link RTT $(i,j)$, bandwidth $BW(i,j)$ and GPU time to produce a chunk $TGPU(i)$. Finetune computes $t'(i) = TGPU(i) + \max RTT(i,j) \sum D(i,j)$. Note that we use a first order model to predict reduction time, instead of reading reduction time. This is because actual reduction time includes overheads from other nodes and we shouldn’t blame node $i$. $t'(i)$ is then divided by $\max t'(i)$ to normalize to a $[0, 1]$ value $r'(i)$. To counteract the effect of imbalance, Finetune also derives $b(i) = \frac{\sum D(i,j)}{\max(\sum D(n,m))}$ and normalized $\hat{b}(i)$.

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 $\frac{\max b(i)}{\min b(i)} > \varepsilon$ (a configuration parameter). If Finetune repeatedly identifies the same bottleneck node, it moves exponentially more LMs and GMs in each step. The number to move is reset to 1 if the bottleneck changes.

Autotune uses a central daemon to collect performance metrics and generates new schedules, and is triggered by a sudden change (larger than 20%) in training performance. Autotune signals AggEngine with the new schedule, which then proceeds with the steps in §4.2.3.

4.4 Integration with Training Frameworks

Integrating PLink with a training framework is straightforward. PLink’s initialization requires only a list of nodes and buffer sizes/addresses from the training framework. Reduction is done by calling reduce in appropriate places. Here we demonstrate how PLink can be integrated with systems with training frameworks with different design decisions.

Caffe2/PyTorch uses blocking reduction. PLink integration is done by extending Gloo [9]’s algorithm object. Caffe2/PyTorch launches multiple algorithm instances and only allows a limited number of concurrent reduce calls. Although PLink can support multi-instances, the concurrency of PLink can be tuned by simply launching more reduction threads and establishing more connections per a pair of nodes. Thus, PLink simply shares the single instance.

Support for MXNet is enabled by extending PSLite [7]’s $\text{kv_dist_server}$ object. MXNet uses an asynchronous callback pattern, but PLink’s reduce is blocking. Thus, a separate event thread is launched for invoking callback through PSLite’s $\text{kv_apps}$ infrastructure when a chunk has finished. Initialization of PLink is done by hooking into the init call of $\text{kv_dist_server}$.

5 Evaluation

We focus our evaluation on three parts of the system and proceed through the following evaluations: (1) Demonstrate the efficiency and scalability of AggEngine as an aggregation engine for distributed training. (2) Quantify the benefits of hierarchical aggregation and topology-awareness in generating a 2LHA plan. (3) Evaluate accuracy of ProbeEmbed’s inference of physical affinity. (4) Assess how well Autotune reacts to network changes. (5) Measure PLink’s impact on end-to-end training performance of popular DL models.

5.1 Environment Setup

We perform our experiments on both Azure and EC2 on various instance types to avoid overfitting. These include NC, F, E and D series on Azure, and P3 and C5 instances on EC2. Each VM runs Ubuntu 18.10 with kernel 4.18, Cuda 9.2 and CuDNN 7.5. We enable accelerated networking [5], or enhanced networking [8] where applicable. All VMs are provided with at least 10 Gbps network throughput.

Some experiments involve comparing results generated with uncertainty. In those cases, we report speedup in expectation with the mean of the samples (>100) we collect, together with percentile metrics and performance distribution data. Each sample represents a potentially different cluster assignment generated by the underlying mechanism (Balanced Random or ProbeEmbed), and each sample collected is the median performance of a 5-iteration reduction latency to minimize the effect of transient network changes. We also interleave runs of different approaches to further reduce the interference from the environment, as suggested in [64].

To find the fastest transport layer for AggEngine, we use a benchmark that measures the median latency of 50 reduction runs of 512MB of data (sufficiently large for throughput measurement), on C5 and Standard D VMs on EC2 and Azure. We found our TCP implementation to be 1.07x and 5.94x faster than SoftiWarp and SoftRoCE implementations. We also found no significant performance differences among various TCP variants. In the following experiments, we use DCTCP [23] as it keeps queuing delays low within a datacenter. To ensure our baselines get the best performance, we enforce all VMs to be in the same availability zone on EC2 and the same datacenter on Azure. § 6 discusses the potential
of PLink to accelerate cross-datacenter, cross-region training.

5.2 Efficiency of AggEngine

This section evaluates the performance of AggEngine itself, disabling ProbeEmbed and Autotune. In particular we examine the performance gains achieved by AggEngine from efficient execution of aggregation schedules. **Comparison to other systems.** For clarity, we first set up our baseline by testing the reduction libraries used in major training frameworks, including MxNet PS-Lite, Nvidia NCCL 2.4, and Facebook Gloo. We use the fastest algorithm (halving-doubling) in Gloo (which is used in Pytorch/Caffe2 [42]). We use all_reduce_perf in NCCL Tests [17] and test_kv_app in PS-Lite for testing their respective performance. Our baseline selection covers implementations of algorithms referenced in §3.2. On both Azure and EC2, we use instances with V100 GPUs and test the end-to-end reduction performance [7] involving copying from/to a GPU. The rest of experiment setup is same as in §5.1.

Figure 6 shows the speedup in terms of median GPU to GPU reduction latency normalized to NCCL’s median latency, when aggregating a large (512MB) and a small (64 byte) buffer. We observe near line rate network utilization with PLink’s flat aggregation (FA) when aggregating large buffers, while other solutions fail to do so. When aggregating a small buffer, algorithms that use fewer rounds (PLink, PS-Lite) have an advantage. PLink FA leads the pack in all settings\(^2\). We now use FA as a strong baseline.

**Scalability.** We also evaluated AggEngine’s scalability by varying the number of participating nodes on Azure, with a buffer size of 512MB. AggEngine achieves a 91% scaling ratio from 2 to 64 nodes. This is because AggEngine’s chunking (§4.2.1) capability keeps per-node workload constant.

\(^2\)PS-Lite does not support end-to-end performance benchmarking with a GPU. Therefore its performance is overestimated.

\(^3\)A non-negligible portion of transfer latency with small buffers comes from GPU operations. AggEngine achieves larger speedup in CPU-only tests.

We now demonstrate additional speedup using various approaches to generate 2LHA schedules, compared to FA, reducing a real-world model (ResNet-50), which contains buffers with different sizes ranging from bytes to megabytes, totalling 97MB. We first present the speedup summary in Figure 7; then, each subsection zooms into details of the benefits associated with individual technique. Overall, PLink is able to achieve up to 3.9x mean and 2.6x P95 speedup compared to FA on EC2, and 1.32x mean and 2.23x P95 speedup on Azure.

5.3.1 Balanced Random-guided 2LHA versus FA

We first show how AggEngine can efficiently run 2LHA without any topology information. We use Balanced Random (§4.1) to generate schedules. Setup is the same as §5.1. Figure 7 shows that Balanced Random, achieves a speedup of 2.4x on EC2 and 1.2x on Azure over FA. Figure 8 zooms into the performance distribution across multiple runs. FA performance is highly volatile due to its sensitivity to network conditions as nodes communicate in an all-to-all pattern. HA performance varies due to variability in the specific clusters generated for each run. However, HA is not only able to achieve a lower expected reduction latency due to its ability to localize transfer, but also a lower tail (P95) latency as its communication is less likely to be disrupted by competing traffic in the datacenter.

To improve our coverage, we ran the same experiments in additional configurations on Azure and EC2 (e.g., using the F, NC and E instances on Azure as well as G3 and P3 instances on EC2), and we observed similar benefits across the different configurations.

5.3.2 ProbeEmbed’s Impact on Hierarchical Aggregation Performance

This section highlights the benefit and necessity of ProbeEmbed, by comparing ProbeEmbed-guided 2LHA with sched-
figure 8: Empirical reduction performance distribution of FA and 2LHA on 64 Azure D32 and EC2 C5n.18xlarge instances. Speedup stats of Balanced Random-guided 2LHA vs FA on Azure and EC2: mean (dotted line): 1.20x and 2.39x. P50: 1.07x and 2.56x. P95: 2.20x and 2.46x.

figures 9: Empirical reduction performance distribution of ProbeEmbed-guided 2LHA vs Balanced Random-guided 2LHA, their embedding graphs, and one instance of k-mean groups generated. Node with same color belong to the same group. Speedup stats: mean: 1.09x and 1.69x. P50: 1.10x and 1.78x. P95: 1.03x and 1.05x.

more stable performance compared to a Balanced Random-based approach by generating more cohesive, locality-preserving group assignments.

Table 7 (column ProbeEmbed) summarizes this. We observe an additional 1.09x and 1.69x of expected performance gain of the clusters generated by ProbeEmbed over those of Balanced Random on Azure and EC2. We present the performance distribution in Figure 9, along with the 2D view of the embedding graph. Notice how VM nodes are laid out in this ring-like embedded plane. ProbeEmbed’s speedup comes from how it can accurately identify groups of VM nodes with locality to exploit, and it generates groups with nodes very close to each other (groups are denoted by the node colors). On the other hand, Balanced Random is likely to generate groups whose intra-group distance is similar to the diameter of the ring, by picking equally-spaced VM nodes on the circumference of the ring, which may result in lower variance but with less-than-ideal performance.

Across all experiments, we found ProbeEmbed-guided 2LHA has a speedup range of 1.04x to 1.5x and 1.02 to 1.25x over random-guided 2LHA on Azure and EC2, respectively. While Balanced Random can generate all possible clusters, ProbeEmbed effectively “cherrypicks” a smaller portion of the partitioning space that better preserves locality, leading to higher performance in expectation. Our experience also indicates larger VM instances often lead ProbeEmbed to have larger speedup gains, likely because they are less likely to be packed into the same physical host, causing VMs to spread across more physical hosts, which results in less inherent locality which Balanced Random is less likely to capture incidentally.

We did not obtain ground-truth information from EC2. However, we did experiments on EC2 that focused on inferring availability zone difference. Our high performance results in these experiments are unremarkable because cross availability zone communication has drastically higher latency.

Comparing with ground-truth-guided 2LHA. We obtain ground-truth topology information from Azure for the VMs that are running⁴, and focus our evaluation on Azure in this section.

The cloud provider can choose to expose topology information to assist tenants in locality-aware scheduling and communication. This section looks at the effectiveness of using ground-truth physical topology to form a reduction schedule for 2LHA. We find our VMs usually span 1 or 2 clusters. We primarily consider grouping VM nodes based on the physical racks they are located in.

We respawn the VMs multiple times to obtain different topologies in order to understand how they affect 2LHA performance. For the runs shown in Figure 7, 2LHA beats ground-truth-guided 2LHA by 1.13x for Azure (which is the setting where we were able to obtain ground-truth data), and these 64 VMs are spread across 22 racks, with the largest rack containing 8 VM nodes while the smallest containing only 1. In two other occasions, these 64 VMs landed in 17 racks and 40 racks respectively, and ProbeEmbed achieves 1.9x and 2.0x speedup over ground-truth-guided approach. We believe 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 (spread over fewer racks), and the more balanced each rack is (similar number of nodes per rack), the better performance of ground-truth-guided 2LHA should be. However, the VM spawn decision 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, like ProbeEmbed does.

Comparing with Balanced Random-guided 2LHA. We now dive deeper into how ProbeEmbed leads to better and
Figure 10: Autotune quickly adapts to each bandwidth changes by rebalancing LM assignments based on current metrics of AggEngine.

Figure 11: Plink’s collective optimizations achieve up to 3x speedup on commercial clouds training popular neural network models, compared to original Pytorch/Caffe2.

5.4 Accuracy of ProbeEmbed

The effectiveness of 2LHA relies on how well ProbeEmbed captures locality. We evaluate three aspects of ProbeEmbed: (1) the accuracy of different types of network probes; (2) the ability of ProbeEmbed to accurately infer VM physical affinity; and (3) the effectiveness of using embedding as a denoising technique.

5.4.1 Definition of Physical Affinity Inference Accuracy

To quantify how accurately ProbeEmbed infers physical affinity, we first define an intuitive metric, the affinity score: 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 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:

\[
A(a, b, c) = \begin{cases} 
1 & \text{if } M(a, c) < M(b, c) \text{ and } T(a, c) < T(b, c) \\
0 & \text{otherwise} 
\end{cases}
\]

For a given observer \(c\), \(A(a, b, c)\) essentially captures whether ProbeEmbed’s measurement agrees with its physical distance to \(a\) and \(b\). Note that \(A(a, b, c)\) is only defined for comparable nodes \(a\) and \(b\) to \(c\). For this, each node’s physical location is encoded as a string of format `region:datacenter:cluster:row:rack:node`. \(a\) and \(b\) are comparable to \(c\) if \(a\) and \(b\) have different longest common prefix to \(c\). Longer common prefix means higher affinity.

We now define affinity accuracy as the total sum of all \(A(a, b, c)\) over the count of all defined \(A(a, b, c)\).s. A higher affinity accuracy represents a better comprehension of the topology of the datacenter network.

5.4.2 Quality of Various ProbeEmbed Probes

We evaluate base affinity accuracy achieved by running just the various measurement probes, without the embedding process. We evaluate these scores on two different instance types on Azure. Table 3 (column DPDK Echo through NTTTCP) shows the raw affinity inference scored. We found our DPDK Echo implementation best captures the network topology. To evaluate with different VM allocations and topologies, we repeat the process of VM deallocation and reallocation multiple times and reprobe each time. We found DPDK’s base affinity score to be the highest, with a range from 0.67 to 1.0.

5.4.3 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. Table 3 (column DPDK Echo + Embedding and Ping + Embedding) shows the boosted accuracy of affinity inference. We observe a significant boost in accuracy for occasions where the base accuracy is less than ideal with DPDK Echo. However, embedding is not a panacea, and it does not help when the base accuracy is too low, which is often the case with ping probes. We also found that the correlation between bandwidth readings and network topology is weak, as bandwidth probes have a similar accuracy as a random guess, and embedding does not help here either. We verified this by observing that probes achieve full line-rate transfers across the datacenter in Azure, albeit at a higher latency. We omit their “boosted” results in the table.

5.5 Effectiveness of Autotune

We continue our evaluation with real world impact of Autotune. We train ResNet-50 with Caffe2 on Standard NC nodes in Azure. We report Autotune’s effects in terms of end-to-end training performance. We do not set a stop condition in the experiment, so Autotune continues to optimize performance. In this experiment, we assess how Autotune reacts to bandwidth changes. We start by imposing no limits on bandwidth, then we limit the bandwidth of node 0, and eventually restore it. With each bandwidth change, we show how the training throughput changes as we apply Autotune decisions.

Figure 10 shows this process. We perform a step of Autotune every 10 iterations to get a stable reading of current throughput. At step=0, node 0’s network throughput is 4 Gbps. At step=1, we limit both upload and download bandwidth of node 0 to 2 Gbps. This causes an immediate drop in training performance and triggers Autotune at step=3. Quicktune moves most GM and LM roles away from node 0. This causes the training throughput to bump up immediately and Autotune enters fine-tuning mode at step=4 until step=20. Finetune achieves a higher training performance compared to no Autotune when node 0 is limited to 2 Gbps, by carefully moving LM assignments from node 0 to 7. When 0 is out of LMs, Finetune moves LMs from 1 to 7. At step=20, node 0’s band-
width is restored, causing an immediate bump in training performance, but this time node 0 is underloaded because it is not serving as LM for any key. At step=21, this is corrected by Quicktune. Autotune then continues to monitor and rebalance workloads throughout, eventually discovering an assignment that leads to slightly better (up to 3%) throughput compared to initial assignment in this experiment.

5.6 End to end training performance

We conclude our evaluation with the impact of PLink’s collective optimizations on the end-to-end training performance of popular neural networks, including ResNet-50, AlexNet and VGG. We present PLink’s speedup compared to original Pytorch/Caffe2\textsuperscript{3} performance, by replacing Gloo with PLink. With each model, we vary the batch size of each GPU to understand how the communication to computation ratio affects PLink’s effectiveness. Figure 11 shows the speedup of PLink, ranging from 1.45x-3.04x on Azure, and 1.04x-1.75x on EC2. Unsurprisingly, PLink shows much larger speedups on models with higher communication to computation ratios than with lower ratios. In particular, training ResNet-50 on 64 VM nodes requires no more than 7 Gbps bandwidth, and speedup for PLink in this case comes from its reduced reduction latency. We also observe, surprisingly, that PLink’s effect is not very sensitive to the batch size chosen.

6 Related Work and Discussion

Cross region training. Many \cite{31, 47} have explored the possibility of geo-distributed learning, where data is not consolidated in one location. PLink can help here both by reducing traffic through bottleneck links (inter-datacenter, inter-availability zone), and by reducing cost of training as cloud providers usually impose a fee on cross region/zone transfers. Large Batch Optimization. One way to alleviate communication bottlenecks is to use a large batch size \cite{16, 42, 52, 83}. Large batch sizes reduce communication frequency. However, that eliminates the potential of achieving a larger speedup with a fast communication plane. For example, with ResNet-50, \cite{78} 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, provided that communication overhead is low. PLink can help in this case.

Quantization, Compression, and Matrix Decomposition. Another orthogonal line of work takes a different approach to solving communication problem by trading potentially more rounds of communication for fewer bytes sent. This can be done with sigma-delta modulation \cite{76}, by decomposing the weight of a fully connected layer \cite{34, 91}, or by reducing redundancy in SGD \cite{60}. These techniques can work well with PLink to take advantage of its lower latency stack.

Reinforcement and Active Learning. An alternate approach for PLink is to leverage recent advancements in reinforcement learning, treating topology inference and continuous rebalancing as a black box with throughput as reward. While past work has shown promising results \cite{29, 32, 33, 56, 68}, our preliminary assessment of end-to-end RL for PLink is unfavourable. RL requires a large sample complexity, and collecting training throughput on commercial cloud has prohibitive costs, as they need to cover variability in the environment (differing topologies between experiments, second-to-second traffic changes, etc). There is also the concern of transfer: an agent is unlikely to see a significant fraction of the possible topologies, and with an always evolving cloud, an agent would need continuous tuning. While this assessment is from the perspective of a cloud customer, a coordinated effort by a cloud provider across customer jobs would be more feasible as it could train on a greater number of samples.

Embedding for Network Awareness \cite{38} takes a related approach to using Euclidean embeddings. We optionally include its “height” parameter inspired by \cite{38} but deviate significantly in our goals. While \cite{38} aims to embed peers on the globe, whereas we use embedding for denoising ProbeEmbed probes with a novel objective.

Predicting Bandwidth Changes. Autotune currently reacts to bandwidth changes only after they happen. This can be improved by carefully monitoring connection state variables such as the congestion window to anticipate an incoming bandwidth change, using techniques as in \cite{26, 45, 55, 59}.

Limitation of HA. HA is most helpful when the bandwidth bottleneck in the system is not in the end-host, which is often the case in the cloud. HA is less helpful in networks with flat hierarchy, where there is less locality to exploit.

7 Conclusion

Accelerating distributed training in the commercial cloud has unique challenges, making communication a bottleneck in the training process. We proposed PLink, a system that accurately probes network and efficiently generates and executes topology-aware, hierarchical aggregation to take advantage of the locality in datacenter network, and continuously balances workload across VM nodes to adapt to changing network conditions. We show that PLink achieves an end-to-end training speedup in unmodified commercial clouds of up to 3x when training popular neural network models.

\textsuperscript{3}Pytorch/Caffe2’s distributed training performance is representative of state-of-the-art today \cite{63}.
References


[25] Vasanth Bala, Jehoshua Bruck, Robert Cypher, Pablo Elustondo, Alex Ho, Ching-Tien Ho, Shlomo Kipnis, and Marc Snir. Ccl: A portable and tunable collective


[75] 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.


