RAT - Resilient Allreduce Tree for Distributed Machine Learning

Xinchen Wan[1] Hong Zhang[2] Hao Wang[1] Shuihai Hu[3]
Junxue Zhang[1] Kai Chen[1,4]

[1]SING Lab @ Hong Kong University of Science and Technology
[2]UC Berkeley [3]Clustar [4]Peng Cheng Lab

ABSTRACT

Parameter/gradient exchange plays an important role in large-scale distributed machine learning (DML). However, prior solutions such as parameter server (PS) or ring-allreduce (Ring) fall short since they are not resilient to issues or uncertainties like oversubscription, congestion or failures that may occur in datacenter networks (DCN).

This paper proposes RAT, a new solution that determines the communication pattern for DML. At its heart, RAT establishes allreduce trees taking into account the physical topology and its oversubscription condition. The allreduce trees specify the aggregation pattern in which each aggregator is responsible for aggregating gradients from all workers within an oversubscribed region at the reduce phase, and broadcasting the updates back to workers at the broadcast phase. We show that such an approach can effectively reduce cross-region traffic and shorten dependency chain compared to prior solutions. We have evaluated RAT in both oversubscribed network and network with failures and found that RAT is resilient to these issues or uncertainties. For example, it delivers an average of 25X and 5.7X speedup compared to PS in oversubscribed network and Ring in network with failures, respectively.

CCS CONCEPTS

• **Computing methodologies → Distributed algorithms**;

KEYWORDS

distributed machine learning, all-reduce algorithm

ACM Reference Format:
Xinchen Wan[1] Hong Zhang[2] Hao Wang[1] Shuihai Hu[3] and Junxue Zhang[1] Kai Chen[1,4]. 2020. RAT - Resilient Allreduce Tree for Distributed Machine Learning. In *4th Asia-Pacific Workshop on Networking (APNet '20), August 3–4, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3411029.3411037

1 INTRODUCTION

Recent years have witnessed an explosive use of deep neural network (DNN) in multiple application domains such as Computer Vision and Natural Language Processing, etc. As DNN training jobs may consume days or weeks to complete, distributed system has been adopted for the purpose of timely training. As a result, we are witnessing tons of researches and approaches involving expediting distributed machine learning (DML) training both in academia and the commercial industry [14, 27, 29, 34].

As a compute-intensive task, DML attracts concentrated efforts to perform efficient cluster scheduling for computation resources. Meanwhile, we detect a shift of performance bottleneck from computation to communication as GPU gets faster and models grow larger [27]. For instance, when training large models such as VGG-16 [31] over 32 GPUs, the communication can take up to 90% of the overall completion time [27]. Substantial approaches have emerged in alleviating the bottleneck in DML, whose purposes vary in network scheduling [13, 18, 29], synchronization mechanisms [16, 22] and communication reduction [23], etc. In this paper we focus on the parameter exchange process in DML.

Parameter exchange schemes describe how parameters are communicated among servers in each iteration. As DNNs are usually trained in 100s to 1000s iterations, there are potential gains for investigating it. PS [21] and Ring [10] are the representative exchange schemes [33] on the market and have been integrated in the mainstream DNN frameworks such as TensorFlow [1], PyTorch [28] and MXNet [6], etc.

APNet '20, August 3–4, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-8876-4/20/08…$15.00
https://doi.org/10.1145/3411029.3411037

Through analysis, however, we reveal that either PS or Ring is essentially static which is topology-agnostic, and not resilient to various issues or uncertainties in datacenter networks (DCN) (see §2). PS adopts a direct communication between workers and servers, which inevitably introduces more cross-rack traffic and creates communication bottlenecks under oversubscription. Meanwhile, Ring adopts chain-like communication pattern that creates extra hop-by-hop dependencies during parameter exchange, making it overly sensitive to events such as congestion or failures. Other alternatives such as k-nominal tree [26, 30], butterfly mixing [20] or recursive halving and doubling [11] can be viewed as an intermediate state between PS and Ring. However, they also suffer from similar problems as PS and Ring to some extent, as they are agnostic to network topology as well. Moreover, some topology-aware schemes [7, 9, 24, 25] have been investigated in recent years, but they either require specific topologies or impose extra hardware requirements (§2).

In response to the aforementioned challenges, we introduce RAT, Resilient Allreduce Tree, as a new parameter exchange scheme with the awareness of physical cluster topology for DML. At its core, RAT resembles the physical topology and establishes allreduce trees over oversubscribed regions (e.g., racks or pods) iteratively. The resultant allreduce trees specify the aggregation pattern in which each aggregator is responsible for aggregating gradients from all workers within an oversubscribed region at the reduce phase, and broadcasting the updates back to workers at the broadcast phase. In this way, RAT effectively minimizes traffic across oversubscribed regions while still maintaining a relatively short communication dependency chain (Table 1), thus being adaptive to network oversubscription and resilient to congestion or failures.

We have evaluated RAT using NS3 simulations with both oversubscribed network and network with failures. Our results show that RAT is resilient to these issues or uncertainties. For example, it delivers an average of 25X and 5.7X speedup compared to PS in oversubscribed network and Ring in network with failures, respectively.

2 BACKGROUND AND MOTIVATION

2.1 Datacenter Networks

DCN usually adopt a multi-layer hierarchical topology [5]. In such topology, switches are connected in a hierarchical way (usually 2 or 3-tier) and servers are grouped by top-of-rack (ToR) switches at leaf level. This kind of topology makes DCN highly flexible for scaling up by simply adding switches at each level and connections between switches and servers.

However, there have been several issues that exist in DCN. Among them include oversubscription, congestion, and failures. Oversubscription is introduced to cut down the high cost of DCN establishment [12]. It leverages the opportunity that all traffic sources are very rarely transmitted at the same time. For a given cluster scale, the number of switches and links can be reduced in this way as compared to 1:1 of the oversubscription ratio. However, oversubscription is a double-edged sword as it brings in a threshold of cluster traffic. While the total traffic exceeds this threshold, congestion can happen at the backbone and in the worst case, break down the whole network. Besides, congestion may occur when bursts flood at certain links or NICs, or when low priority flows may be starved by the high priorities on switches, etc. Failures can take place at the physical layer, for example, physical link failures, nodes failures and so on.

2.2 Distributed Machine Learning

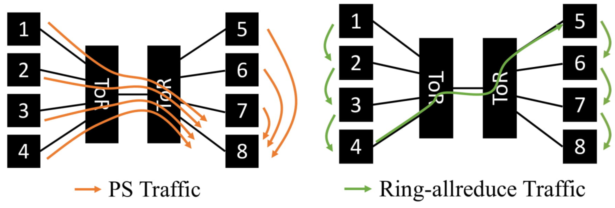
Generally, parallelism schemes of DML can be categorized as data parallelism and model parallelism, and data parallelism is the most prevailing option. In such paradigm, each worker manages its local model and trains independently on a portion of the dataset. The training process is done in an iterative way, and each iteration contains two phases. The first phase is the compute-intensive local model training phase that involves a forward pass to generate predictions with the mini-batch input and a backward pass to derive local gradients with respect to the loss between predictions and the given labels. The second phase is the communication-intensive parameter exchange phase, where mean gradients are calculated across all locally calculated gradients. Updated parameters are sent back to each worker, and the workers start the next iteration with the updated version.

The parameter exchange phase described above typically follows a Bulk Synchronous Parallel (BSP) synchronization mode, which is already the most prevalent synchronization in production because of its best ML tasks' performance and reproducibility. In this mode, all workers are barriered in each iteration and the new iteration cannot start until all workers have finished updating their local models in the current iteration.

2.3 Drawbacks of Existing Schemes

At each parameter exchange phase, a particular exchange scheme that describes the logical parameter exchange process among servers in each iteration is implemented. Here we categorize popular parameter exchange schemes for DML jobs, and discuss each of their limitations as our motivation to infer the desired properties of DML and yet to design an efficient parameter exchange mechanism:

- PS: Adopted by several DNN frameworks like TensorFlow [1], Caffe [19] and MXNet [19]. PS employs a direct communication pattern where parameters are synchronized directly between workers and PSes. After computing and



(a) PS

(b) Ring-allreduce

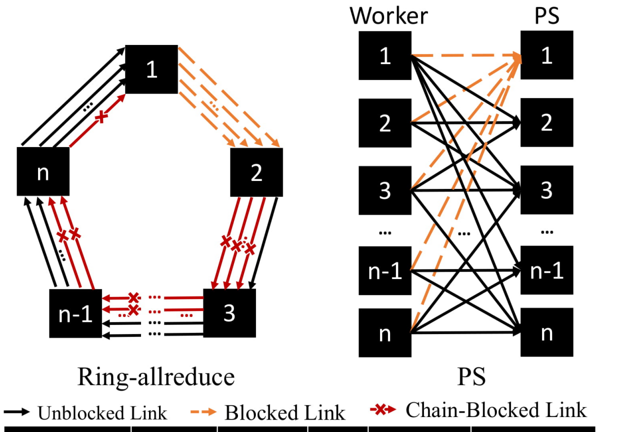
Figure 1: Traffic pattern of PS and Ring

generating local gradient updates, workers *push* them directly to PSes and *pull* the updated parameters back as soon as PSes finish the aggregation.

Though PS is direct yet effective, we reveal that it is **ill-suited to network oversubscription**. A PS traffic pattern example is shown in Figure 1(a). Assuming workers and PSes are co-located in each of the nodes, we observe that the cross-rack link will always suffer 16/7x workload compared to each intra-rack link. More generally, given $|r|$ racks each with $|w_r|$ workers with an oversubscription ratio of o, the average ACT will be at least $\frac{o\cdot|w_r|-1}{|w_r|-1/|r|}$ times worse. This means that the problem will get more severe for large jobs with more racks, and our experiment in §4 also validates this inference. Note that the server placement in each rack won't alleviate this problem, because the inter-rack traffic won't change given a cluster size. And the key factor is the direct communication pattern which is adopted by PS.

- Ring-allreduce: Used in BaiduRing [10] and Horovod [30]. All nodes form a ring topology, and each node transmits gradients in exactly the same circular direction. It has two phases: *scatter-reduce* and *all-gather*. In the *scatter-reduce* phase, after generating gradient updates, each worker receives a chunk of gradients from its left-hand side (anti-clockwise for example), aggregates it with its local copy, and send it to its right peer. After $n-1$ iterations, each worker has precisely one chunk that involves all workers' updates. Then in the *all-gather* phase, each of the n workers simply copy the received chunk with $n-1$ iterations, and then complete the communication phase.

  Compared to PS, Ring-allreduce minimizes the inter-rack traffic by aggregating the parameters in each hop (see Table 1). But it introduces too many dependencies, and is thus **vulnerable to congestion or failures**. As the case shown in Figure 2, n nodes are involved in implementing allreduce operations, and we assume node 1 cannot send data temporally, which may result from multiple reasons such as the link fails, or is congested and priority is given to other traffic, or it is a straggler and not yet ready to send, etc. In this case, node 2 can only send $1/n$ of the data to node 3 via one of its chains, because $n-1$ chains are blocked due to node 1. In turn, node 3 can only send $2/n$ of the data to node 4, and so on. Such dependency causes a cascading effect to all the downstream nodes, leading to a 50% cutoff on network utilization when n is large. We refer to such phenomenon as chain blocking, and our simulation result in §4 shows its impact. In contrast, PS does not suffer from this problem as direct communication introduces minimal dependency.



Network Util	1	2	…	N	Avg. (n->inf)
Ring	0	1/n	…	1-1/n	50%
PS	0	(n-1)/n	…	(n-1)/n	100%

Figure 2: Ring allreduce suffers from chain blocking

- Other collective allreduce schemes: Other allreduce schemes like k-nominal tree [26], butterfly mixing [20] and recursive halving and doubling [11] can be viewed as a mixture of PS and Ring. They have predetermined exchange patterns that are agnostic to network topologies, and suffer from similar problems like extra traffic for inter-rack communication and long dependencies to some extent. We list each of their corresponding values in Table 1 and emphasize their limitations.
- Topology-aware allreduce schemes: Some recent allreduce schemes [7, 9, 24, 25] execute gradient aggregations by awaring the hierarchical network topology, but they each more or less face problems in the context of large scale network. BlueConnect [7] breaks up one ring into multiple small rings with the awareness of the network topology. It works in a more fine-grained manner and alleviates the impact caused by the slowest link of the ring. However, as it's a variant of ring-based scheme, it inherits the vulnerability of Ring and would run worse when each rack scales up. HiPS [9] embraces RDMA transport for allreduce and works specifically for server-centric network topology, but would introduce extra dependency

chain when it runs in ring mode. ParameterHub [24] works as a parameter exchange scheme that co-designs both software and hardware. At its core, PBoxes (a server equipped with 10 NICs) are used within ToRs to reduce cross-rack traffic. However, it introduces a special preference of extra hardware (multiple NICs for aggregation) and cannot guarantee the minimal cross-region traffic. PLink [25] applies a 2-level hierarchical aggregation upon the topology, but have the same issue of extra traffic when the hierarchy exceeds 2.

3 DESIGN

The limitations discussed in §2.3 inspires us of the desired properties of the logical parameter exchange scheme:

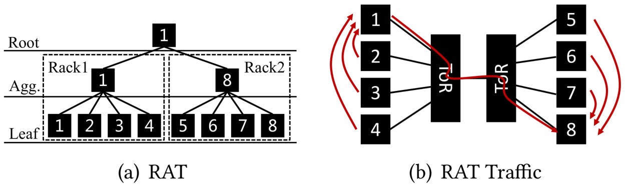
- *Minimum traffic across oversubscribed regions* (e.g., rack, pod) to avoid in-network bottlenecks;
- *Short dependency chain* for better resilience to traffic congestion and failures;
- *Simple structure* to enable timely embedding with affordable computation and enforcement overhead.

We proceed to introduce RAT, a topology-aware parameter exchange scheme with the following parts: its key roles that match network topology, the algorithm that describes the establishment of each allreduce trees, and reveal that its properties are accord with the desired ones.

Key roles of RAT: Given a physical network topology **T**, we build the logical RAT for a DML job **J** following a simple layered structure, resembling the physical topology while accounting for the oversubscribed regions (i.e., racks, pods). A node plays one or more of the following roles:

- **Leaf:** sends its local gradients and receives the global update. Each worker in job **J** corresponds to a leaf.
- **Aggregator:** for each oversubscribed region in the topology **T**, RAT introduces a corresponding aggregation layer to minimize cross-region traffic. In the reduce phase, an aggregator aggregates the gradient updates from leaves and lower level aggregators within the region, and send the aggregated updates to the higher level aggregator or root. In the broadcast phase, the communication reverses.
- **Root:** aggregates all the gradients, calculates the global update, and sends it back in the reversed direction.

Algorithm of RAT: The general algorithm of RAT for establishing allreduce trees is described in Algorithm 1. RAT partitions the whole nodes into different groups and hierarchically aggregates gradients based on the topological characteristics. The aggregation process operates in this way: at the bottom leaf layer, a level-0 aggregator is assigned to each physical rack, a.k.a oversubscribed regions, and takes charge of aggregating all gradients within the same racks. Thereafter, a level-1 aggregator is designated from all level-0 aggregators and aggregates gradients among level-1. The aggregations at higher levels follow the same routine, until all gradients originated at leaves are aggregated in one single level-(n-1) aggregator, which is also known as the root. Afterward, the broadcast operation starts and operates hierarchically in the reversed direction.



(a) RAT (b) RAT Traffic

Figure 3: The RAT with the topology in Figure 1 and its traffic pattern

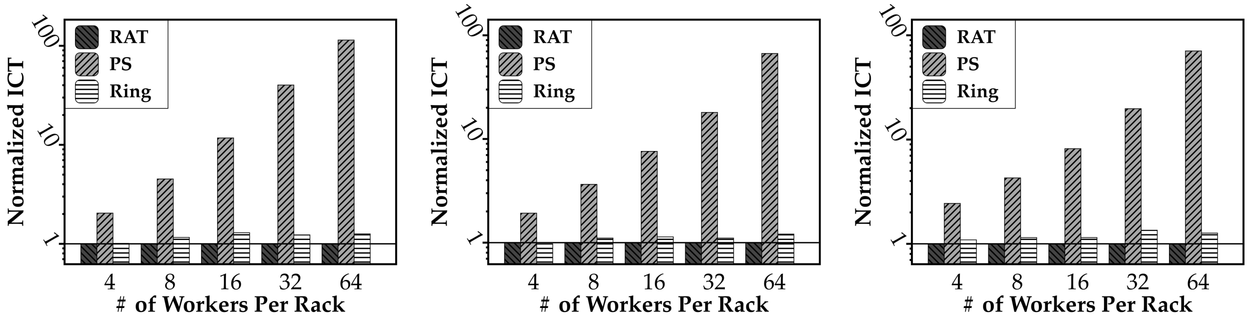
Algorithm 1: RAT Algorithm

```
Input:
$h$: The number of hierarchical levels
$n_i$: The number of groups in $ith$ level
$G$: The total gradients to synchronize by this process
$group_{ij}\_array$: The array of node_ids in the $jth$ group
in level $i$ of topology $T$
begin
    $Agg\_array = []$
    for $i \leftarrow 0$ to $h - 1$ do
        for $j \leftarrow 0$ to $n_i - 1$ do
            Set $N_{ij} \leftarrow Len(group_{ij}\_array)$
            Random pick integer $m$ from $[0, N_{ij}]$
             Set $Aggregator \leftarrow group_{ij}\_array[m]$
            append $Aggregator$ to $Agg\_array$
            Reduce($Aggregator, G, group_{ij}\_array$)
        end
    end
    Set $k \leftarrow Len(Agg\_array)$
    for $i \leftarrow h - 1$ to $0$ do
        for $j \leftarrow n_i - 1$ to $0$ do
            $Agg\_id = Agg\_array[k - 1]$
            Set $N_{ij} \leftarrow Len(group_{ij}\_array)$
            Set $Aggregator \leftarrow Agg\_id$
            Broadcast($Aggregator, G, group_{ij}\_array$)
            $k- = 1$
        end
    end
end
```

As an example, Figure 3 shows a RAT given the network topology of 8 nodes in 2 racks. Each worker corresponds to a leaf, and an aggregator/root can be designated to any

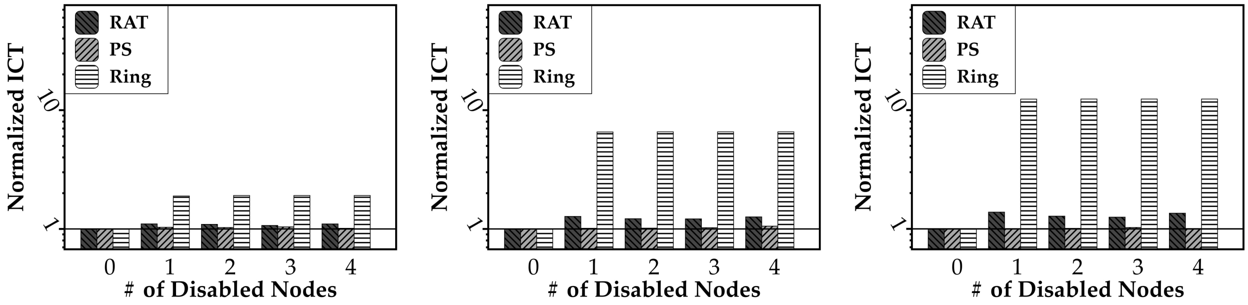
Schemes	PS	Ring	Butterfly	Halving &doubling	K-nominal tree	BlueConnect	PLink	RAT
Minimum cross domain traffic	×	√	×	×	×	√	×	√
No. of Dependant steps	2	$2(\lvert w\rvert - 1)$	$\log_2(\lvert w\rvert)$	$2\log_2(\lvert w\rvert)$	$2\log_k(\lvert w\rvert)$	$2(\lvert l\rvert + max\{w_r\})$	4	$2(\lvert l\rvert + 1)$

Table 1: RAT achieves good tradeoff minimizing cross-region traffic and length of dependency.



(a) **10Gbps** (b) **40Gbps** (c) **100Gbps**

Figure 4: RAT's speedup in an oversubscribed scenario



(a) **d=1ms** (b) **d=5ms** (c) **d=10ms**

Figure 5: RAT's speedup in a network with failures scenario

worker and executes the aggregation task within the same oversubscribed region. Note that we only consider rack-level oversubscription/aggregator in our analysis hereafter for simplicity. Besides, as we can form a total of 32 RATs in this topology, we uniformly distribute the traffic on each RAT for load balancing because we assume our network is symmetric and follow the routine that each RAT carries equal workload. For the context of the asymmetric network topology, we leave it as future work for exploration.

Properties of RAT: Table 1 shows how RAT achieves the desired properties by comparing it against the alternative parameter exchange schemes. Note that l is the number of oversubscription layers, w represents the total worker number, and w_r refers to the number of workers in each rack.

Now we show that RAT satisfies the desired properties. First, we observe that all alternative solutions except Ring and BlueConnect cannot minimize the traffic cross oversubscribed regions. In contrast, RAT is tailored for the physical topology, which optimizes this by introducing an aggregator for each oversubscribed region. Second, RAT introduces a $2(|l| + 1)$ dependency chain. Since a datacenter cluster typically has few (say 1 or 2) oversubscription layers, this chain is usually much smaller than alternative patterns except PS. Third, RAT follows a simple and regular structure with 3 different roles, thus greatly simplifying the computation and enforcement (§4) of the parameter exchange process.

4 EVALUATION

In this section, we run simulations to quantify the high utilization of RAT by comparing with two representative parameter exchange schemes, PS and Ring.

4.1 Simulation Setup

Experimental settings: We use two different experimental settings in our simulation. In the oversubscribed scenario, we use a conventional spine-leaf topology with 2 spines switches and 4 leaf switches. We set the # of workers per rack as the variant and the oversubscription ratio changes accordingly (from 2:1 to 32:1). In the network with failures scenario, we run ML traffic on 64 servers (under 2 racks) in a 40G network with no oversubscription. We simulate the network congestion or failures at nodes or links by pausing some nodes sending data. That is, we randomly select k nodes temporarily stop sending data, and periodically change the k nodes every d time. Note that the training performance is measured as each job's iteration completion time (ICT).

Traffic: We simulate the traffic pattern of PS, Ring and RAT in NS3. For PS, we set PSes and workers co-locate with each other and simulate the process as all-to-all sending equal size of data simultaneously. For RAT, we construct # of RATs and assign each workers the root role in each RAT symmetrically. We distribute the total traffic uniformly on each RATs for load balancing. And in the Ring case, we connect all nodes in ring mode logically and allow them to communicate with neighbors. We simulate the network traffic the same size as ResNet50 [15] (97MB in total) and distribute it uniformly upon the three cases. Note that for simplicity, we assume that there is no overlap between computation and communication. Though the result may be inaccurate when tensors are small, we claim that the inaccuracy is limited by recalling that network-intensive models experience large skewness at the tensor size.

4.2 Results

Oversubscribed Scenario: As shown in Figure 4, PS performs 25X worse than RAT under all bandwidth settings because it introduces a large amount of inter-rack traffic and results in the bottleneck at cross-rack link. Ring minimizes inter-rack traffic and is expected to perform as well as RAT. However, from the figure we surprisingly observe a 0.16x throughput degradation in ring implementation in many cases. Through analysis, we think Ring's long dependency chain may introduce some extra delay at each hop. The delay at each hop slows down the whole training process.

Network with Failures Scenario: We also demonstrate the stability of RAT in the network with failures scenario. As described above, we create a network with failures environment in our topology and deploy a distributed DML job on it. Note that to show the performance degradation when there exist disabled nodes in network, we normalize the results with the $k = 0$ case.

The result is shown in Figure 5. Ring suffers from much severe slowdown (with an average of 12x degradation in the worst case) compared to PS and RAT, which is consistent with our analysis in §2. When one node is blocked, other nodes can still take the available bandwidth to proceed if in PS or RAT mode. While for Ring, the progress of its downstream nodes is also heavily blocked due to chain blocking. As different nodes can be blocked at different times, a node in the Ring can always be blocked - either by itself or by some upstream nodes. In comparison, RAT achieves comparable performance to PS since it only introduces a minimal number of additional dependencies upon PS (2 in this case).

5 RELATED WORK

Optimizing DML: There exist many solutions which can be used to optimize the communication for DML. For example, techniques such as gradient compression [23] and quantization [2] can transfer fewer or compressed gradients for communication acceleration. Solutions like layer-wise communication scheduling and prioritization [13, 18, 29] can maximize the interleave of computation and communication. Furthermore, traditional approaches that minimize network flow completion time by using flow scheduling [3, 4] or coflow scheduling [8, 32, 35] can also be leveraged for DML communication optimization. All these solutions are orthogonal to RAT, and RAT can cooperate with these techniques by further alleviating the bottleneck of cross-region links.

Cross region training: Some works [17] explore approaches for geo-distributed training. They alleviate the impact of latency across regions by reducing dependencies and allow the use of stale parameters. However, these approaches sacrifice the reproducibility of ML jobs and may affect the final job performance. RAT is designed to reduce traffic across oversubscribed regions but keeping the widely used BSP for the promise of reproducibility and great job performance.

6 CONCLUSION

This paper presented RAT, a new parameter exchange solution with topology awareness for DML. At its heart, RAT establishes allreduce trees by considering the physical topology characteristics and the trees form a hierarchical pattern in which each aggregator aggregates gradients from all workers within an oversubscribed region at the reduce phase, and broadcasts the updates back to workers at the broadcast phase. RAT achieves both minimal cross-region traffic and short dependency chain goals compared to prior parameter exchange schemes. We have simulated RAT in NS3 and our results demonstrated the potential of RAT: it delivers an average of 25X and 5.7X speedup compared to PS in oversubscribed network and Ring in network with failures, respectively.

REFERENCES

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *OSDI 2016*.

[2] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2017. QSGD: Communication-efficient SGD via gradient quantization and encoding. In *NIPS 2017*.

[3] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pfabric: Minimal near-optimal datacenter transport. In *SIGCOMM 2013*.

[4] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. 2015. Information-agnostic flow scheduling for commodity data centers. In *NSDI 2015*.

[5] Kashif Bilal, Samee Ullah Khan, Joanna Kolodziej, Limin Zhang, Khizar Hayat, Sajjad Ahmad Madani, Nasro Min-Allah, Lizhe Wang, and Dan Chen. [n. d.]. A Comparative Study Of Data Center Network Architectures.

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

[7] Minsik Cho, Ulrich Finkler, and David Kung. 2019. BlueConnect: Novel Hierarchical All-Reduce on Multi-tired Network for Deep Learning. In *SysML 2019*.

[8] Mosharaf Chowdhury and Ion Stoica. 2015. Efficient Coflow Scheduling Without Prior Knowledge. In *SIGCOMM 2015*.

[9] Jinkun Geng, Dan Li, Yang Cheng, Shuai Wang, and Junfeng Li. 2018. HiPS: Hierarchical parameter synchronization in large-scale distributed machine learning. In *Proceedings of the Workshop on Network Meets AI & ML 2018*.

[10] Andrew Gibiansky. 2017. Bringing HPC techniques to deep learning. *Baidu Research, Tech. Rep.* (2017).

[11] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).

[12] Albert Greenberg, James Hamilton, David A Maltz, and Parveen Patel. 2008. The cost of a cloud: research problems in data center networks. (2008).

[13] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H Campbell. 2018. TicTac: Accelerating distributed deep learning with communication scheduling. *arXiv preprint arXiv:1803.03288* (2018).

[14] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. 2018. Applied machine learning at facebook: A datacenter infrastructure perspective. In *HPCA 2018*.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR 2016*.

[16] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. 2013. More effective distributed ml via a stale synchronous parallel parameter server. In *NIPS 2013*.

[17] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R Ganger, Phillip B Gibbons, and Onur Mutlu. 2017. Gaia: Geo-distributed machine learning approaching {LAN} speeds. In *NSDI 2017*.

[18] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. 2019. Priority-based parameter propagation for distributed DNN training. In *SysML 2019*.

[19] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093* (2014).

[20] John Kim, William J Dally, and Dennis Abts. 2017. Flattened butterfly: a cost-efficient topology for high-radix networks. In *ISCA 2007*.

[21] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *OSDI 2014*.

[22] Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. 2015. Asynchronous parallel stochastic gradient for nonconvex optimization. In *NIPS 2015*.

[23] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. 2018. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *ICLR 2018*.

[24] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. 2018. Parameter hub: a rack-scale parameter server for distributed deep neural network training. In *SOCC 2018*.

[25] Liang Luo, Peter West, Jacob Nelson, Arvind Krishnamurthy, and Luis Ceze. 2020. PLink: Efficient Cloud-based Training with Topology-aware Dynamic Hierarchical Aggregation. In *SysML 2020*.

[26] Luo Mai, Chuntao Hong, and Paolo Costa. 2015. Optimizing network performance in distributed machine learning. In *HotCloud 2015*.

[27] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *SOSP 2019*.

[28] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in pytorch. (2017).

[29] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A generic communication scheduler for distributed DNN training acceleration. In *SOSP 2019*.

[30] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[31] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[32] Hengky Susanto, Hao Jin, and Kai Chen. 2016. Stream: Decentralized Opportunistic Inter-Coflows Scheduling for Datacenter Networks. In *IEEE International Conference on Network Protocols (ICNP) 2016*.

[33] Songtao Wang, Dan Li, Yang Cheng, Jinkun Geng, Yanshu Wang, Shuai Wang, Shu-Tao Xia, and Jianping Wu. 2018. Bml: A high-performance, low-cost gradient synchronization algorithm for dml training. In *NIPS 2018*.

[34] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. 2018. Gandiva: Introspective cluster scheduling for deep learning. In *OSDI 2018*.

[35] Hong Zhang, Li Chen, Bairen Yi, Kai Chen, Mosharaf Chowdhury, and Yanhui Geng. 2016. CODA: Toward Automatically Identifying and Scheduling Coflows in the Dark. In *SIGCOMM 2016*.