Characterizing Distributed Deep Learning Training Performance in Datacenters

Paras Jain, Sam Kumar, Alexander Mao*

ABSTRACT

Distributed deep learning enables training larger, and thus more accurate, models. However, as clusters grow larger, communication can come to dominate runtime. The network must be carefully balanced with the training hardware to avoid one bottlenecking the other. However, the current practice of distributed deep learning is to schedule nodes without consideration for their job topology. In our work, we investigate the impact of the network on training speed. Through careful construction of a representative distributed DNN training cluster, we simulate the impact of various scheduling policies on overall performance. We find that network-aware job placement can significantly impact DNN training time; compared to a random-assignment baseline, a best-case placement strategy improves performance by about 25% and an adversarial placement strategy reduces performance by 200-300%. By adjusting the latency distribution of computation in our simulations, we confirm that communication over the network is indeed a significant bottleneck in DNN training. We believe this demonstrates an achievable opportunity gap and outline future directions in topology-aware scheduling and in-network aggregation.

1 INTRODUCTION

Deep learning has enabled state-of-the-art performance on common computer vision tasks such as image classification [12, 13, 15, 22, 24] and in natural language processing [2, 10, 28, 29, 42]. However, deep neural networks (DNNs) have an unquenchable thirst for compute—training ResNet-50 [15] to full accuracy requires 4.65 exaflops and can take over a week to complete on a single node.

As researchers design models in an iterative process, it is important to reduce the time to train a single model. Deep learning training can be distributed across a cluster of nodes in order to reduce the time to train a model. Facebook trains DNN models across a cluster of 256 GPUs [14], and Tencent has trained a DNN across a cluster of 2048 GPUs [19].

In order to distribute DNN training across several nodes, Stochastic Gradient Descent (SGD) is often used. SGD is an algorithm for iteratively finding the optimal parameters for a model under some differentiable loss objective, and it enables parallel optimization via data parallelism where each node trains the DNN on a local shard of the data.

During this process, there are three phases: (a) the *forward pass* where each node trains the model on a local shard of data, (b) the *backward pass* where each node calculates the gradient, or the update to the parameters, and (c) *parameter reduction* where each node averages its local copy of training results, or the gradient, with all other nodes. The forward and backwards pass are both computation-heavy operations but parameter reduction is dominated by communication and thus bottlenecked by the network.

The parameter server [9, 25, 26] approach is a popular gradient aggegation architecture where all nodes communicate with a single node that computes the accumulation. However, as the number of training nodes increases, the ratio of computation to communication decreases dramatically (communication scales superlinearly to number of nodes but computation remains constant [40]). As a result of Amdahl's law, communication is the bottleneck in state-of-the-art large-scale DNN training algorithms (see Figure 1 for an explanation for why). Accelerators like GPUs, TPUs, and FPGAs make communication even more of a bottleneck.

Our aim is to study how task placement in a cluster affects DNN training performance. In our ongoing preliminary work, we measure the impact of the network on distributed deep learning. Specifically, we explore several parameters of a distributed deep learning setup under the parameter server model. This work establishes a lower bound on the speedup from smarter job placement in datacenters. We consider the impact of *colocating* nodes in a single training job. This reduces contention on core routers which are often underprovisioned relative to edge routers [1]. We find network topology-aware scheduling can have a significant impact on DNN training cost, with a 3x difference between the best and worst case job scheduling policies.

Our ongoing work's key contributions include:

(1) We present detailed profiles of a real deep learning cluster that realistically model the computation phases of distributed deep learning.
(2) We present a packet-level simulation of a deep learning cluster and compare the current practice in the public cloud against a network-aware scheduling policy.
(3) We discuss opportunities for accelerated communication in distributed deep learning clusters.

*Authors listed in alphabetical order

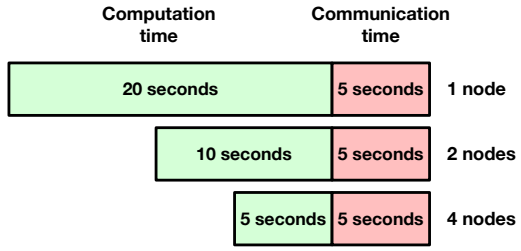

Figure 1: Distributing SGD reduces per-node gradient latency. However, communication does not scale. Thus, communication bottlenecks large-scale training.

Algorithm 1: Synchronous Stochastic Gradient Descent with Parameter Server

for *epoch* $n \leftarrow 1$ **to** N **do**
  for *minibatch* $d \subset |D|$ **do**
    for *worker* $w \leftarrow 1$ **to** W **do**
      PULL(θ_{t-1})
      $\hat{y} \leftarrow f_\theta(d_w, \theta_{t-1})$
      $\nabla_w \leftarrow \nabla_\theta L(y, \hat{y})$
      PUSH(∇_w)
    end
    Barrier on M PUSH operations
    $\theta_t \leftarrow \theta_{t-1} - \eta \sum_{w=1}^{W} \nabla_w$
  end
end

2 BACKGROUND AND RELATED WORK

Deep learning has exploded as a field over the last few years. It is the core method behind advances in computer vision [12, 13, 15, 22, 24], language processing [2, 10, 28, 29, 42] and even systems [11, 18, 21]. Distributed deep learning has been a recent area of interest in the systems community as the hefty computational requirements of deep learning presents notable challenges across the stack.

2.1 Deep learning optimization

Deep learning encompasses a family of machine learning models that automatically learn representations [17, 23] of data, in either a supervised (*i.e.* $y = f_\theta(x)$) or unsupervised (*i.e.* $p_\theta(x)$) settings. Deep neural networks are defined by the composition of many layers that are typically some kind of non-linear function. Each layer is parameterized by a set of learned parameters or weights θ_i. A loss function $L(y,\ \hat{y})$ defines the objective a practitioner would like to minimize. The iterative process of finding the optimal parameters by optimizing $\arg\min_\theta \sum^i L(y,\ \hat{y})$ is referred to as *training*.

Stochastic Gradient Descent [20, 35] (SGD) is an optimization method and is the most common algorithm used for training DNNs. SGD iteratively the parameters θ using first-order gradient information (specifically $\theta_t = \theta_{t-1} - \eta \nabla_\theta L(y,\ \hat{y})$). SGD typically will make multiple passes over the dataset, and batches training over small portions of the data called a mini-batch. Evaluating the neural network's prediction $\hat{y}$ is referred to as the forward pass while computing the gradient $\nabla_\theta L(y,\ \hat{y})$ is the backwards pass.

2.2 Distributed DNN training

Training on large datasets can be scaled up by distributing training across multiple nodes. Data-parallelism distributes training with each node calculating gradients locally on their own shard of the data, followed by gradient aggregation. There are two common forms of distributed SGD — asynchronous and synchronous.

In asynchronous SGD, each worker often will update the gradients in arbitrary order [9, 31]. This potentially means updating the shared set of parameters using a gradient computed on a stale copy of the parameters. The staleness of parameters can be bounded to improve convergence [25].

Synchronous SGD includes a lock to block parameter updates until all workers have computed the results of their local mini-batch [5]. Synchronous SGD will produce the exact same result as a larger effective mini-batch size. In this work, we consider synchronous SGD as it maintains statistical convergence guarentees.

2.3 Parameter server

In both setups of distributed SGD, parameters are often synchronized using a parameter server. While first applied to sparse logistic regression [25], the parameter server represents a simple way to synchronize gradient updates in both synchronous and asynchronous settings. Parameters are stored on a single (or sharded [9]) master node which all workers communicate with for both retrieval and updates. Communication is simply modeled by an interface where clients can PUSH or PULL parameters from a single server. The full algorithm is described in detail in algorithm 1.

2.4 Optimizing gradient reduction

Various hand-optimized variants of the parameter server have been designed in order to improve network throughput. Optimus [33] empirically optimizes the number of servers in relation to the number of clients. Paramter Hub [27] optimizes the placement of parameter servers relative to rack-level topology in networks. However, Parameter Hub is a single design that is not flexible across diverse datacenter network topologies like Clos [1, 38] and Jellyfish [39].

The parameter server pattern can create hotspots in the network via incast [6]. Methods like AllReduce [32, 45] reduce redundant traffic sent in the network through collective communication patterns which have no single master node and have been implemented in numerous libraries [30, 37, 43]. BlueConnect [7] and GradientFlow [40] both decompose the AllReduce operation into hierarchical reduction to maximally utilize the network.

Optimus [33] and Gandiva [44] attempt to schedule distributed deep learning jobs with various heuristics to improve cluster utilization and throughput. We consider job colocation in this work as a heuristic.

3 PROBLEM STATEMENT AND SCOPE

Given that large-scale distributed DNN training jobs tend to exhibit communication bottlenecks, a natural research objective is to optimize communication in DNN training. As mentioned in the previous section, much of the existing work to optimize communication bottlenecks in DNN training focuses on the *algorithm to distribute the computation*. In this report, we explore a different approach to optimize the communication bottleneck in DNN training. **Our high-level goal is to explore to what extent optimizing the underlying network has potential to alleviate the communication bottleneck in distributed DNN training.**

3.1 Possible Approaches

There are a variety of approaches to exploring this goal:
Optimizing network topology for communication patterns. One natural approach is to *build the network topology according to the communication patterns required by the underlying algorithm that distributes DNN training*. For example, if the parameter server algorithm is used, one may consider connecting the parameter server to each of the workers directly via dedicated high-bandwidth low-latency links. If tree AllReduce is used, one could connect the compute nodes in a tree-shaped network that mirrors the communication pattern. If ring AllReduce is used, one could consider placing the compute nodes in a ring or torus-shaped topology.
Optimizing communication patterns for network topology. A variant of the above approach is for compute nodes to receive information about their relative positions within the network, and use it to chose a *placement-specific communication pattern*. The plan for aggregating gradient updates across different workers could be chosen according to network placement to minimize communication overhead.
Hardware communication support. Yet another line of work takes advantage of network hardware to eliminate communication bottlenecks. Network hardware has the potential to accelerate deep learning by (1) optimizing the communication directly, and (2) performing computation in the network directly. For example, one approach is to use RDMA [34, 40] to achieve more predictable latency for communication. Another possibility, explored by concurrent work [36], considers using programmable switches [3, 4] for aggregation.

3.2 Our Approach

Our goal in this project is to determine the *potential* of network optimization strategies to improve the performance of distributed DNN training. While existing work using programmable switches has shown up to a 300% speedup [36], approaches relying on specialized hardware and network configurations (e.g., network topologies specialized for DNN training) are more brittle than software-only solutions. We elected to focus on solutions that can be deployed without hardware changes.

In particular, we focused on measuring the potential of *intelligent, network-aware job placement* to accelerate DNN training. Specifically, we compare DNN training performance with different job placement strategies to characterize its performance with (1) best-case placement, (2) average case placement, and (3) worst-case placement. Our purpose in doing so is two-fold. First, we would like to understand the relative importance of avoiding pathologies in job placement and optimizing beyond the average case. Second, we seek to understand the potential of hardware-free network-based approaches to optimize the communication bottleneck in DNN training. A positive result would indicate that DNN training could be optimized at the network level without brittle hardware-based strategies; a negative result would indicate that network-based approaches cannot appreciably improve DNN training performance.

Among algorithms to distribute DNN training performance, we focused on the parameter server algorithm. Alternatives to parameter server include AllReduce-based strategies, designed to improve communication performance. However, we chose to focus on parameter server because it provides the biggest opportunity for network optimization. Furthermore, the presence of network-level optimization might change the algorithm-level changes that yield better performance, making us hesitant to use an AllReduce algorithm optimized for naïve network placement.

4 METHODOLOGY

We used the ns-3 network simulator [16] to estimate the efficiency of DNN training under various job placement strategies. We set up a topology in ns-3 resembling a datacenter network, and varied the placement of jobs in the network.

4.1 Network Topology

We design our network topology in ns-3 around a rack abstraction. We model a rack as a set of hosts connected to a

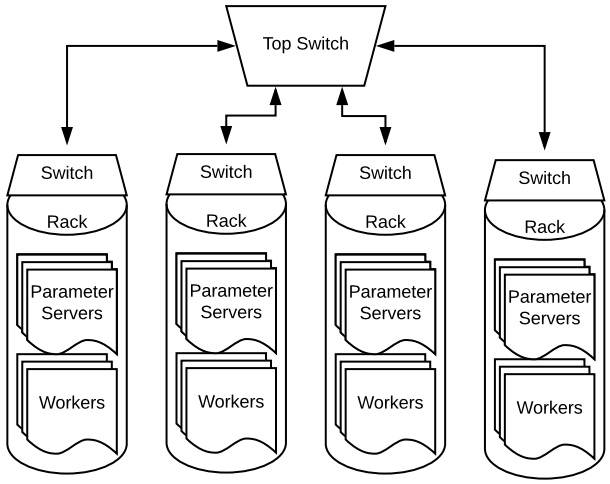

Figure 2: Network topology in ns-3

single top-of-rack switch. We henceforth refer to this top-of-rack switch as a *rack switch*. Our overall topology consists of multiple racks, where each rack switch is connected to a central switch we call a *top switch*. Our topology is illustrated in Figure 2.

Real datacenter topologies are often structured as a fat tree [1, 38]. In our topology, this would correspond to having multiple top switches, each connected to all rack switches, and then using connection hashing (e.g., ECMP) to distribute TCP flows across the multiple paths between racks. However, implementing multiple top switches, to the point of achieving full bisection bandwidth, is not realistic, as real datacenters typically have more bandwidth within a rack than they do across racks [8]. We felt it was important to model this trend, since overprovisioned bandwidth within each rack could potentially make intelligent job placement less effective. Given that rack switches were connected to the top switch via a depth-1 hierarchy, we opted to have a single top switch, which guaranteed a single unique path between any two rack switches (and therefore, between any two hosts). If we had a deeper hierarchy connecting rack switches, we would have had to support multiple paths between racks to support full bisection bandwidth among rack switches (though we would still overprovision bandwidth within a single rack).

We implemented rack switches using ns-3's `bridge` module, but found it easiest to implement the top switch as an IPv4 router with statically computed routes. We do not expect that this affected the results of our simulation because ns-3 does not, to our knowledge, simulate processing delays when transferring packets from one interface to another.

4.2 Simulating the Parameter Server

We focused on the parameter server algorithm (Section 2.3) for distributing DNN training in our cluster. Communication overhead in distributed computation is often exacerbated by *stragglers* [41]; each iteration of SGD is outsourced to multiple worker nodes, but the iteration time is determined by the *slowest* worker. Properly modeling stragglers is essential to obtain meaningful results—if most of the communication overhead is due to stragglers in computation, any optimization to the network can have limited impact, at best. Unfortunately, ns-3 is not the right tool to model computation of gradient updates on the GPU, and the associated stragglers—ns-3 is designed to model packet dynamics, not computation.

In order to model the delays in (1) computing gradient updates on worker nodes, and (2) updating model parameters on the parameter server, we empirically measured the latency distributions for performing this computation on Amazon EC2 when training ResNet-50 [15]. We measured the time to compute a gradient update on the GPU at a worker node (p3.16xlarge instance) and the time to aggregate gradient updates on the CPU at the parameter server. We measured the size of gradients to be 97.49 MB.

Based on our empirical measurements of the time to calculate these values (Figure 6 and Figure 7), we implemented the parameter server and workers in ns-3 as applications with the following behavior:

(1) Parameter server sends a block of data, the same size as a model update, to each of its workers.
(2) Once a worker receives its model update, it samples from the latency distribution for computing gradient updates, and waits for that amount of time.
(3) Each worker sends a block of data, the same size as a gradient update, to the parameter server.
(4) Once the parameter server receives gradient updates from all workers, it samples from the latency distribution for computing model updates, and waits for that amount of time.
(5) Go back to Step 1.

Running the above steps once represents a single iteration of SGD. Our performance metric is the latency distribution for running a single iteration of SGD. Training a complete model involves executing SGD for a certain number of iterations, depending on the exact model being trained and data it is being trained on. Because optimizing the network affects only the speed of each iteration, not the results, it does not affect the number of iterations required. Thus, *the total training time is directly proportional to our performance metric, namely the latency of a single SGD iteration.*

4.3 Job Placement

Section 4.1 and Section 4.2 described our setup and performance metric. Now, we describe the parameter we vary, namely *job placement*, to study how it affects the performance metric.

We refer to a parameter server and its workers collectively as a *job*. Measuring a single training job in isolation would not be realistic, because it would have exclusive access to the entire network. Real-world network performance of DNN training would be affected by *cross-traffic*, namely competition for buffer space from other jobs running in the datacenter network. To properly simulate cross-traffic, we run *multiple concurrent but independent DNN training jobs in the network*.

We consider four job placement strategies, which we describe in turn below:

(1) `Colocated.` This represents the best-case job placement: each job runs in its own dedicated rack. Because there is no communication across jobs (all communication is between nodes in a single job), no traffic is sent through the top switch.
(2) `Random.` This represents the status quo: each job's nodes are assigned randomly in the network, without any attention to colocating nodes.
(3) `Strided.` This represents the opposite of the *colocated* case; each job's worker nodes are placed on different racks than its parameter server. However, each parameter server is placed on a separate rack. Therefore, all traffic must flow through the top switch, but no rack sees more traffic than any other.
(4) `Clustered.` This represents the worst-case job placement. It is the same as the *strided* case, except that all parameter servers are placed in the same rack. Hence, all traffic must flow through the top switch, and all traffic will flow through a single link, namely the link connecting the top switch to the rack switch for the rack containing all parameter servers.

We illustrate these job placement strategies (except `random`) in Figure 3, Figure 4, and Figure 5, where each cylinder represents a rack in our topology.

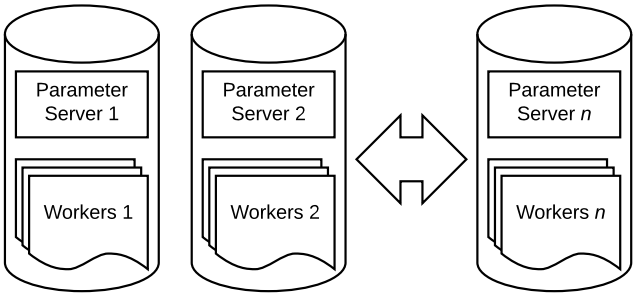

Figure 3: Colocated job placement

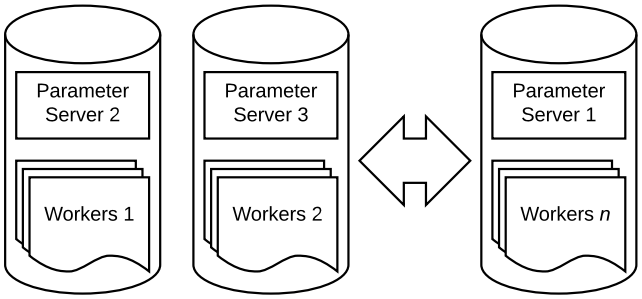

Figure 4: Strided job placement

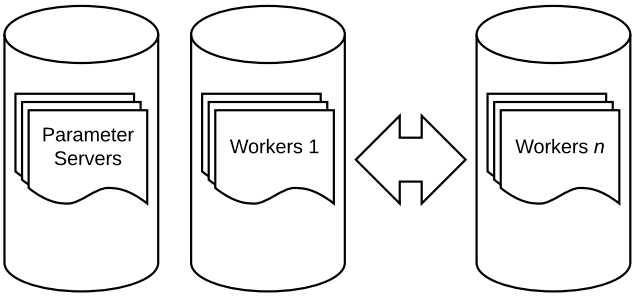

Figure 5: Clustered job placement

4.4 Implementation

We implemented the above functionality in ns-3 as a C++ simulation script with applications and helpers, also in C++. Overall, our implementation is about 1000 lines of C++.

We initialized our topology with 8 racks, each containing 8 servers. Each job consisted of a parameter server and 7 other nodes (total of 8 nodes per job), and we scheduled 8 jobs in the topology at a time according to the different job placement policies described in Section 4.3. We set the bandwidth of each link in our topology to have a bandwidth of 10 Gbps and a latency of 15 ns (with CSMA as in an Ethernet link). However, we found that our simulations took a long time to run, because ns-3 simulates each packet separately when transferring a 97.49 MB gradient vector. Therefore, we scaled down the simulation by a factor of 1000—we decreased the size of the gradient update to 97.49 KB, and comparably decreased the link bandwidth to 10 Mbps so that the overall transfer time would not be significantly affected.

5 EVALUATION

Our main goal in the evaluation is to determine the potential of hardware-free network optimization strategies to improve DNN training performance.

5.1 Modeling distributed DNN training

As mentioned in Section 4.2, we used *empirical measurements* of computation latency distributions in order to capture the effect of stragglers on DNN training performance. Figure 6 shows the latency distribution of computing gradient updates on worker nodes, and Figure 7 shows the latency distribution of applying these gradient updates at the parameter server. The key takeaway from these distributions is that they are *heavy-tailed*, as expected—a significant amount of probability mass is much higher than the median. Therefore, we do expect stragglers to affect DNN training performance.

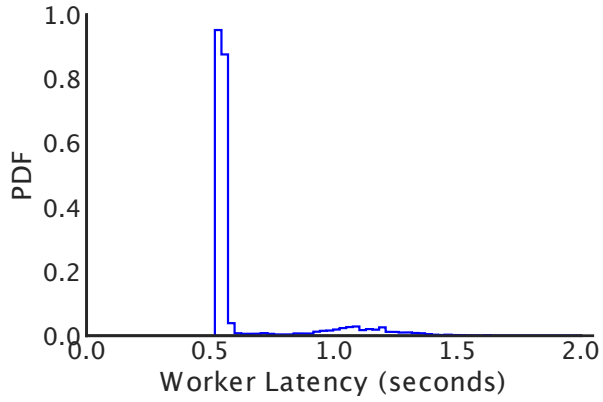

Figure 6: Empirical distibution for gradient computation times on each worker's GPU. A few samples were distributed out to 8s, but are not included in this histogram.

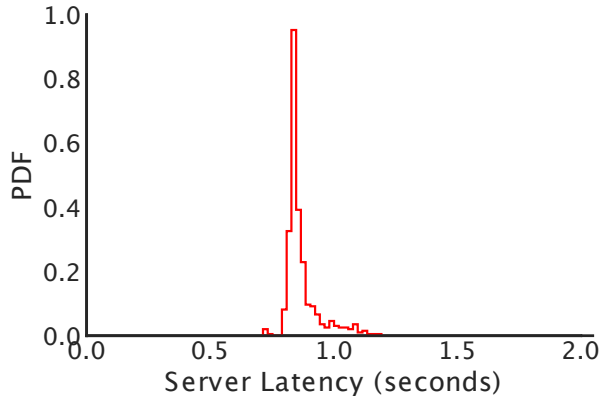

Figure 7: Empirical distibution for server gradient reduction time performed on CPU.

5.2 SGD Iteration Latency with Heavy-Tailed Computation

In our initial set of simulations, we modeled the computation delays at the workers and parameter servers by sampling from the empirically-determined heavy-tailed latency distributions shown in Figure 6 and Figure 7. We ran these simulations on each of the four placement strategies described in Section 4.3, and measured the latency distribution of completing a single SGD iteration. The results for the four placement strategies are shown in Figure 8.

The `colocate` job placement strategy resulted in the best performance. This was to be expected since it represents a "best-case" layout: it requires no communication across racks. However, it provides only a small improvement of about 25% compared to the `random` layout, which represents network-agnostic job placement.

The `cluster` job placement strategy, which represents an adversarial "worst-case" layout, performs 2-3x worse than the other placement strategies. In particular, there is a larger difference between the `cluster` strategy and the `random` baseline than there is between the `random` baseline and the `colocate` strategy. **This suggests that it is more important to avoid pathologies in job placement (i.e., avoid the worst case) than it is to optimize performance beyond the average case.**

The `stride` placement strategy performed similarly to the `random` baseline, in that both have similar median SGD iteration latencies. The `stride` policy, however, has a significantly smaller variance in latency compared to the `random` baseline; in this sense, `stride` outperformed `random`. Despite putting each job's workers on a different rack from its parameter server, `stride` performed very well, significantly outperforming the `cluster` pathology and almost matching

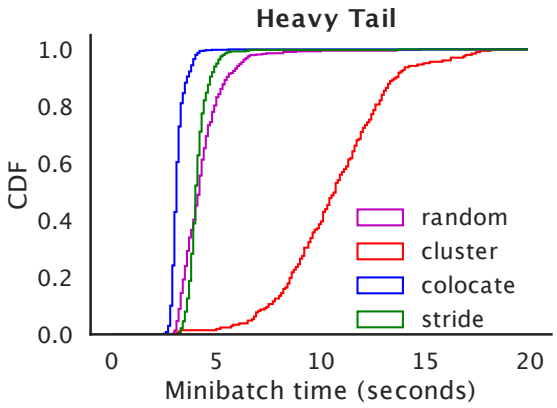

Figure 8: CDF of SGD iteration latency for various job placement strategies (using empirical latency distribution for computation)

the best-case `colocate` policy. **This suggests that optimizing *across jobs* to avoid contention for a single rack is more important than colocating nodes *within a job*.**

5.3 SGD Iteration Latency with Normally-Distributed Computation

In Section 5.2, we used the heavy-tailed latency distribution we empirically measured to simulate computation delays and

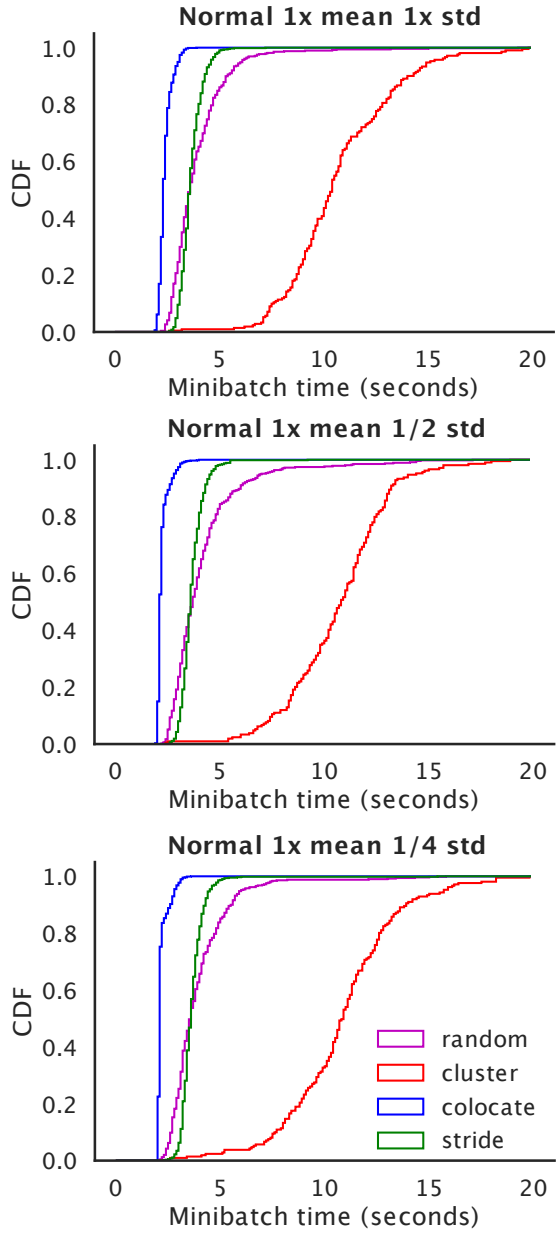

Figure 9: *What if workers were more consistent?* Empirical CDF of SGD iteration latency where computation latency is sampled from a normal distribution. We vary the standard deviation of the normal distribution across the different graphs.

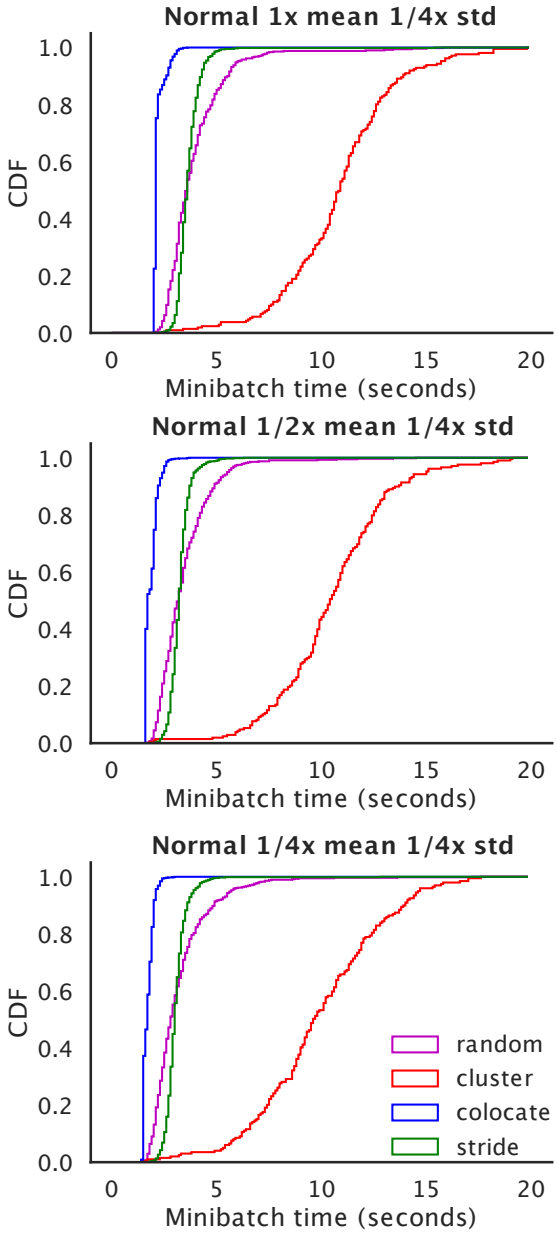

Figure 10: *What if hardware were faster?* Empirical CDF of SGD iteration latency where latency is sampled from a normal distribution. We vary the mean of the normal distribution across the different graphs. Note that the baseline is the same as in Figure 9.

stragglers. However, stragglers limit the potential of optimizing the network, due to Amdahl's Law; even if the network's overhead were zero, DNN training would be limited by stragglers that cause training to block waiting for them. A natural question is: *if systems researchers and engineers manage to decrease the variance in the computation latency distribution, thereby decreasing the impact of stragglers, would that increase the potential of optimizing the network to improve DNN training performance?*

To explore this question, we modified our simulation to use a normal distribution to simulate the computation delay, rather than the heavy-tailed distributions we empirically measured. The mean and standard deviation of the normal distribution, denoted μ and σ, were set to be the same as those of the empirical distribution, whose mean and standard deviation we denote as μ_0 and σ_0.

Comparing the first graph in Figure 9, which models computation latency as a normal distribution, to Figure 8, which models computation latency as our empirical heavy-tailed distribution, we can see that moving from a heavy-tailed distribution to a normal distribution slightly decreases the latency of SGD iterations, but not significantly so.

We also decreased the standard deviation used in the normal distribution to $\sigma = \frac{1}{2}\sigma_0$ and $\sigma = \frac{1}{4}\sigma_0$ (while maintaining $\mu = \mu_0$) in the other two graphs in Figure 9, to see how further decreasing the variance in computation time might affect the relative performance. As expected, this made the distribution smoother. There is, however, still quite a heavy tail in our distributions suggesting that much of the variance in our results comes from the network, rather than the variance in worker gradient calculations and parameter server gradient aggregation latencies.

We also decreased the mean used in the normal distribution to $\mu = \frac{1}{2}\mu_0$ and $\mu = \frac{1}{4}\mu_0$ (while maintaining $\sigma = \frac{1}{4}\sigma_0$) to see whether faster computation hardware might potentially make the network more of a bottleneck, as predicted by Amdahl's Law. The results are shown in Figure 10. Decreasing the mean computation time did decrease the overall iteration time, but not appreciably so. This confirms that the network is indeed a significant bottleneck already. It is also consistent with our expectations based on theoretical calculations: sending eight single gradient updates ($\approx$ 100 MB) over a 10 Gb link to the parameter server would take more than half a second in the ideal case, and would probably take about a second once other overheads (e.g., lack of synchronization, TCP overhead) are taken into account. This lower bound cannot be overcome by improving computation speed or even optimizing job placement.

6 CONCLUSION

Distributed deep learning training is computationally heavy and often bottlenecked by communication overheads. How much impact does the network have on DNN training? This paper investigated whether intelligent job placement has the potential to improve DNN performance by alleviating communication bottlenecks. We performed a series of ns-3 simulations to investigate this. Our findings are as follows:

- Avoiding pathologies in job placement has greater potential to improve DNN training performance than optimizing placement beyond the average case.
- Optimizing *across* jobs to avoid hot spots in the network is more important than colocating nodes *within* a job. For the parameter server algorithm in particular, it is important to avoid placing parameter servers from different jobs in the same part of the network, as they may compete for limited bandwidth.
- Algorithm-level and system-level innovations to make computation faster or more predictable would improve DNN training performance, but not significantly so. This suggests that the network is indeed an important bottleneck, and that optimizing the network (as opposed to computation) has potential to improve DNN training performance.

The opportunity gap outlined in this paper motivates future work into network-topology aware gradient reduction. The ns-3 based model outlined in this paper can be extended to ring or tree AllReduce patterns. These measurement results indicate the potential for a smart logical-to-physical scheduler that minimizes network congestion in multi-tenant deep learning clusters. Moreover, specialized hardware for in-network gradient reduction can reduce the volume of packets sent through the network. Finally, multicast may improve communication throughput in collective communication primitives.

REFERENCES

[1] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. [n. d.]. A Scalable, Commodity Data Center Network Architecture. ([n. d.]), 12.

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural Machine Translation by Jointly Learning to Align and Translate. *arXiv:1409.0473 [cs, stat]* (Sept. 2014). http://arxiv.org/abs/1409.0473 arXiv: 1409.0473.

[3] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. [n. d.]. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. ([n. d.]), 12.

[4] Pat Bosshart, George Varghese, David Walker, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, and Amin Vahdat. 2014. P4: programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (July 2014), 87–95. https://doi.org/10.1145/2656877.2656890

[5] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. 2016. Revisiting Distributed Synchronous SGD. *arXiv:1604.00981 [cs]* (April 2016). http://arxiv.org/abs/1604.00981 arXiv: 1604.00981.

[6] Yanpei Chen, Rean Griffith, Junda Liu, Randy H. Katz, and Anthony D. Joseph. 2009. Understanding TCP Incast Throughput Collapse in Datacenter Networks. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking (WREN '09)*. ACM, New York, NY, USA, 73–82. https://doi.org/10.1145/1592681.1592693 event-place: Barcelona, Spain.

[7] Minsik Cho, Ulrich Finkler, David Kung, and Hillery Hunter. [n. d.]. BlueConnect: Decomposing All-Reduce for Deep Learning on Heterogeneous Network Hierarchy. ([n. d.]), 11.

[8] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, et al. 2018. Andromeda: performance, isolation, and velocity at scale in cloud network virtualization. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 373–387.

[9] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marcaurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. 2012. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 1223–1231. http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv:1810.04805 [cs]* (Oct. 2018). http://arxiv.org/abs/1810.04805 arXiv: 1810.04805.

[11] Jim Gao. 2014. Machine learning applications for data center optimization. (2014).

[12] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014. Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation. In *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition (CVPR '14)*. IEEE Computer Society, Washington, DC, USA, 580–587. https://doi.org/10.1109/CVPR.2014.81

[13] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative Adversarial Nets. In *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 2672–2680. http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf

[14] Priya Goyal, Piotr DollÃąr, 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. (June 2017). https://arxiv.org/abs/1706.02677v2

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, Las Vegas, NV, USA, 770–778. https://doi.org/10.1109/CVPR.2016.90

[16] Thomas R Henderson, Mathieu Lacage, George F Riley, Craig Dowell, and Joseph Kopena. 2008. Network simulations with the ns-3 simulator. *SIGCOMM demonstration* 14, 14 (2008), 527.

[17] Geoffrey E Hinton. 1984. Distributed representations. (1984).

[18] Nathan Jay, Noga H. Rotman, P. Brighten Godfrey, Michael Schapira, and Aviv Tamar. 2018. Internet Congestion Control via Deep Reinforcement Learning. *arXiv:1810.03259 [cs]* (Oct. 2018). http://arxiv.org/abs/1810.03259 arXiv: 1810.03259.

[19] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, Tiegang Chen, Guangxiao Hu, Shaohuai Shi, and Xiaowen Chu. 2018. Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes. *arXiv:1807.11205 [cs, stat]* (July 2018). http://arxiv.org/abs/1807.11205 arXiv: 1807.11205.

[20] J. Kiefer and J. Wolfowitz. 1952. Stochastic Estimation of the Maximum of a Regression Function. *The Annals of Mathematical Statistics* 23, 3 (Sept. 1952), 462–466. https://doi.org/10.1214/aoms/1177729392

[21] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. 2018. Learning to Optimize Join Queries With Deep Reinforcement Learning. *arXiv:1808.03196 [cs]* (Aug. 2018). http://arxiv.org/abs/1808.03196 arXiv: 1808.03196.

[22] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 1097–1105. http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[23] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (Nov. 1998), 2278–2324. https://doi.org/10.1109/5.726791

[24] Christian Ledig, Lucas Theis, Ferenc Huszar, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, and Wenzhe Shi. 2017. Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, Honolulu, HI, 105–114. https://doi.org/10.1109/CVPR.2017.19

[25] 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. 583–598. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu

[26] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. 2014. Communication Efficient Distributed Machine Learning with the Parameter Server. In *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 19–27.

[27] 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. *arXiv:1805.07891 [cs]* (May 2018). http://arxiv.org/abs/1805.07891 arXiv: 1805.07891.

[28] Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective Approaches to Attention-based Neural Machine Translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Lisbon, Portugal, 1412–1421. https://doi.org/10.18653/v1/D15-1166

[29] Thang Luong, Ilya Sutskever, Quoc Le, Oriol Vinyals, and Wojciech Zaremba. 2015. Addressing the Rare Word Problem in Neural Machine Translation. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, Beijing, China, 11–19. https://doi.org/10.3115/v1/P15-1002

[30] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, and Ion Stoica. [n. d.]. Ray: A Distributed Framework for Emerging AI Applications. ([n. d.]), 19.

[31] Feng Niu, Benjamin Recht, Christopher Re, and Stephen J. Wright. 2011. HOGWILD!: A Lock-free Approach to Parallelizing Stochastic Gradient Descent. In *Proceedings of the 24th International Conference on Neural Information Processing Systems (NIPS'11)*. Curran Associates Inc., USA, 693–701. http://dl.acm.org/citation.cfm?id=2986459.2986537

event-place: Granada, Spain.

[32] Pitch Patarasuk and Xin Yuan. 2009. Bandwidth optimal all-reduce algorithms for clusters of workstations. *J. Parallel and Distrib. Comput.* 69, 2 (Feb. 2009), 117–124. https://doi.org/10.1016/j.jpdc.2008.09.002

[33] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. ACM, New York, NY, USA, 3:1–3:14. https://doi.org/10.1145/3190508.3190517 event-place: Porto, Portugal.

[34] Y. Ren, X. Wu, L. Zhang, Y. Wang, W. Zhang, Z. Wang, M. Hack, and S. Jiang. 2017. iRDMA: Efficient Use of RDMA in Distributed Deep Learning Systems. In *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 231–238. https://doi.org/10.1109/HPCC-SmartCity-DSS.2017.30

[35] Herbert Robbins and Sutton Monro. 1951. A Stochastic Approximation Method. *The Annals of Mathematical Statistics* 22, 3 (1951), 400–407. https://www.jstor.org/stable/2236626

[36] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter Richtárik. 2019. Scaling Distributed Machine Learning with In-Network Aggregation. *CoRR* abs/1903.06701 (2019). arXiv:1903.06701 http://arxiv.org/abs/1903.06701

[37] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv:1802.05799 [cs, stat]* (Feb. 2018). http://arxiv.org/abs/1802.05799 arXiv: 1802.05799.

[38] Arjun Singh, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs HÃűlzle, Stephen Stuart, Amin Vahdat, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, and Bob Felderman. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication - SIGCOMM '15*. ACM Press, London, United Kingdom, 183–197. https://doi.org/10.1145/2785956.2787508

[39] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P Brighten Godfrey. [n. d.]. JellyïňĄsh: Networking Data Centers Randomly. ([n. d.]), 14.

[40] Peng Sun, Wansen Feng, Ruobing Han, Shengen Yan, and Yonggang Wen. 2019. Optimizing Network Performance for Distributed DNN Training on GPU Clusters: ImageNet/AlexNet Training in 1.5 Minutes. *arXiv:1902.06855 [cs]* (Feb. 2019). http://arxiv.org/abs/1902.06855 arXiv: 1902.06855.

[41] T. v. Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. 1992. Active Messages: A Mechanism for Integrated Communication and Computation. In *[1992] Proceedings the 19th Annual International Symposium on Computer Architecture*. 256–266. https://doi.org/10.1109/ISCA.1992.753322

[42] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, ÅĄukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 5998–6008. http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf

[43] Cliff Woolley. [n. d.]. NCCL: ACCELERATED MULTI-GPU COLLECTIVE COMMUNICATIONS. ([n. d.]), 56.

[44] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. [n. d.]. Gandiva: Introspective Cluster Scheduling for Deep Learning. ([n. d.]), 17.

[45] Huasha Zhao and John Canny. 2013. Butterfly Mixing: Accelerating Incremental-Update Algorithms on Clusters. In *Proceedings of the 2013 SIAM International Conference on Data Mining*. Society for Industrial and Applied Mathematics, 785–793. https://doi.org/10.1137/1.9781611972832.87