Dissecting the Communication Latency in Distributed Deep Sparse Learning

Heng Pan[†] [‡], Zhenyu Li[†] [‡], JianBo Dong[§], Zheng Cao[§], Tao Lan[§], Di Zhang[§], Gareth Tyson[♭], Gaogang Xie[*]

[†]ICT, CAS, China [‡] Purple Mountain Laboratories [§]Alibaba Group [♭]QMUL, United Kingdom
[*]CNIC, CAS, China

ABSTRACT

Distributed deep learning (DDL) uses a cluster of servers to train models in parallel. This has been applied to a multiplicity of problems, *e.g.* online advertisement, friend recommendations. However, the distribution of training means that the communication network becomes a key component in system performance. In this paper, we measure the Alibaba's DDL system, with a focus on understanding the bottlenecks introduced by the network. Our key finding is that the communications overhead has a surprisingly large impact on performance. To explore this, we analyse latency logs of 1.38M Remote Procedure Calls between servers during model training for two real applications of high-dimensional sparse data. We reveal the major contributors of the latency, including concurrent write/read operations of different connections and network connection management. We further observe a skewed distribution of update frequency for individual parameters, motivating us to propose using in-network computation capacity to offload server tasks.

CCS CONCEPTS

• **Networks → Network performance analysis**;

ACM Reference Format:
Heng Pan[†] [‡], Zhenyu Li[†] [‡], JianBo Dong[§], Zheng Cao[§], Tao Lan[§], Di Zhang[§], Gareth Tyson[♭], Gaogang Xie[*]. 2020. Dissecting the Communication Latency in Distributed Deep Sparse Learning. In *ACM Internet Measurement Conference (IMC '20), October 27–29, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3419394.3423637

1 INTRODUCTION

Deep learning (DL) is used by a range of Internet applications, *e.g.* computer vision, natural language processing, and speech recognition. With the increasing sophistication of algorithms and the volume of data, the size of DL models have become very large. This is particularly extreme for models that involve large and high-dimensional sparse data, such as recommendation systems, online advertising and search engine. For example, the Alibaba advertising system generates petabytes (PB) of user behavior logs everyday, and the training samples from this data contain billions of features, of which only a few are non-zero for each sample [12].

To train such large models, *distributed deep learning* (DDL) has emerged as a common practice. It leverages a large-scale cluster of servers (called workers), each equipped with one or more GPUs, to perform the training task cooperatively. Thus, the most common way to scale out and accelerate DL training is parallelisation. Nevertheless, the performance is often sub-linear with the scale of the cluster. To highlight this, we show in Figure 1 the training time reduction when varying the number of workers in Alibaba's DDL platform and training a real online search engine (see Section 2.2) in asynchronous mode. Here, the Alibaba's DDL platform follows the parameter server (PS) architecture [14], where each worker is assigned almost equal-sized data for training. Specifically, we vary the scale of the cluster from 32 to 512 (*i.e.* # of workers), denoted by n, to perform the same training task, and evaluate their training latency ('experiment'). For comparative purposes, we train the same neural network model on a small fixed-scale cluster that consists of only 16 workers but with $1/x$ of the whole training data, where x is $n/16$, and also record its training latency ('baseline'). As a result, no matter in the experiment or baseline, the workload of each worker is identical (*i.e.* $1/n$ of the whole training data). We observe a notice gap between the baseline and experiment. This is caused by the communication overhead, which lowers the speedup factor by 50% when 512 workers are used for parallel training. This result is echoed by recent literature [19].

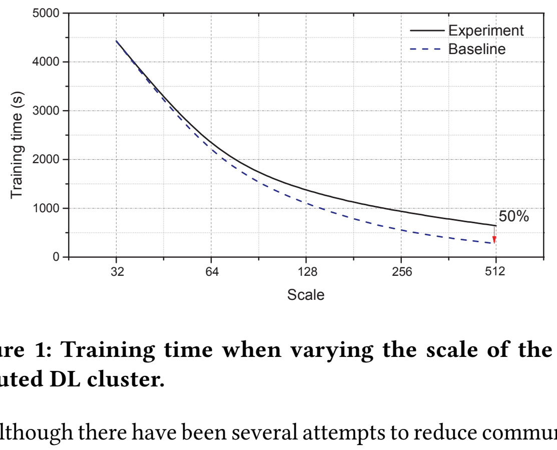


Figure 1: Training time when varying the scale of the distributed DL cluster.

Although there have been several attempts to reduce communication overhead through scheduling algorithms [11, 19, 23], data transmission compression [5, 26] or new network technologies [4, 16], we still lack a deep understanding of DDL platforms. Similarly, compared to conventional distributed computation systems (*e.g.* Hadoop [1] and Spark [17]) or a distributed query system [6], the communication patterns of DDL platforms are distinct, *e.g.* periodic communications per iteration and long-lived connections during training. Thus, we argue it is vital to better analyse DDL to develop targeted improvements.

To this end, we present a measurement study of the communication latency of Alibaba's DDL system. We instrumented a system to gather breakdown latency data for each Remote Procedure Call (RPC) during the training of two applications: advertising recommendations and their search engine. Our dataset contains latency logs for 1.38M RPC calls. We also analyse the number of concurrent connections to individual PS nodes and the parameter update frequencies during the training process. With this data, we first investigate the dominant contributors to latency, and then examine possible bottlenecks. Based on our findings, we discuss the potential of using in-network computation to mitigate the bottlenecks. Note that Alibaba's DDL platform follows the popular PS architecture. We thus believe our measurement results are generalizable to other systems using the similar architecture for sparse data training, and can be used to direct the optimizations of other DDL platforms.

To summarize, we make the following three contributions:

- We find that the communication overhead significantly lowers the performance of the training process. A breakdown analysis reveals that the parameter server (PS) processing time dominates the latency for both local parameter updating (from workers to PS nodes) and global model synchronisation (from PS nodes to workers). Network transmission time (including the network stack time at both sides) is another major contributor to delay.
- We identify that large transmitted messages, frequent communications and concurrent active connections are the three key causes of the high PS processing time and network transmission time. Furthermore, increasing the number of workers while keeping the ratio of workers-to-PS nodes results in an increased PS CPU utilisation, because of the increased burden on connection management and read/write competitions among connections.
- We find a skewed distribution of update frequency of individual parameters. The top 20,000 *hot* parameters that are updated most frequently account for 50% and 70% of total updates for recommendation and search applications, respectively. We propose to use the computation capacity of programmable switches to aggregate hot parameters to alleviate the read/write burden of PS nodes.

2 BACKGROUND AND DATASET

In this section, we present a brief overview of DDL, as well as our industrial dataset that drives our analysis.

2.1 Alibaba's DDL Platform

Alibaba's DDL platform adopts the parameter server (PS) architecture [14], which uses *data parallelization* to perform DDL training in an asynchronous mode (see Figure 2). In data parallelization mode, the training data is divided into equal-sized parts, each of which is assigned to a training node (a.k.a worker) to start its local training. The platform uses RDMA (Remote Direct Memory Access) [9] for underlying data transmission, and machines are connected by 10Gbit networks in a LAN.

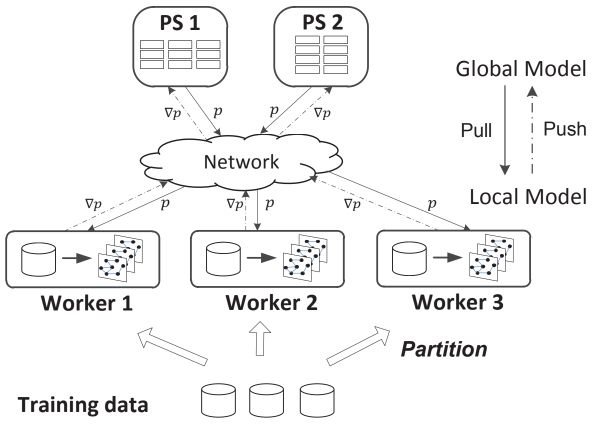


Figure 2: Overview of parameter server architecture.

A PS node is in charge of a set of parameters that do not overlap with those of other PS nodes. A worker maintains a long-lived reliable connection to each of the PS nodes during training. In each iteration, a worker reads data samples to form a mini-batch and *pulls* the updated model parameters that it will use in this integration from the corresponding PS nodes. After finishing the local training (which may take a relative long time, *i.e.* a long interval), the worker will *push* the locally computed gradients back to PS nodes for aggregation and parameter updates. This marks the end of an iteration on this node. The node will then read a new mini-batch for the next iteration of training. Note that pipelining may be used to accelerate the above process [12]. Furthermore, a *pull*/*push* operation may involve several RPC (Remote Procedure Call) messages, each of which contains part of the data that needs to be pulled from or pushed to the PS node.

2.2 Datasets

To explore the nature and bottlenecks of DDL, we rely on a dataset collected from Alibaba's DDL system, which trains models for high-dimensional sparse data. Note that, different from XDL which uses two neural networks for one application [12], the examined DDL system trains one neural network (a.k.a one model) per application. The system consists of 20 PS nodes and 80 workers.

We have 3 types of data: (*i*) processing latency for each RPC; (*ii*) concurrent *active* connections at individual PS nodes; and (*iii*) the statistics on parameter update frequency. Our datasets cover the whole training process for two applications: recommendation system (R1) and search engine (R2). Both involve high-dimensional sparse data and are very challenging for DDL [12]. Table 1 summarizes the data that we used in this paper. It presents the number of logs we have for the latency statistics and the concurrent active connections; as well as the number of parameters in each model. The rest of this section describes the three types of data in detail.

Table 1: Summary of used datasets.

App.	#lat. logs	# conn. logs	# parameters
R1	480 k	130 k	0.9 m
R2	900 k	200 k	150 m

Latency data . A log is generated for each RPC at the worker that initiates the call. It contains: the worker ID, PS node ID, iteration number, a flag indicating push or pull, the timestamp marking the sending time (in μs), the size of RPC request (in bytes), the size of the response (in bytes) and five latency metrics (in μs) that together measure the time overhead of this RPC (see Figure 3). These latency metrics are measured by instrumenting the training programs at both worker and PS sides. Note that it is not necessary to synchronize all machine clocks. During the experiment, we calculate the difference only between two timestamps that are recorded on the same machine. The metrics are as follows:

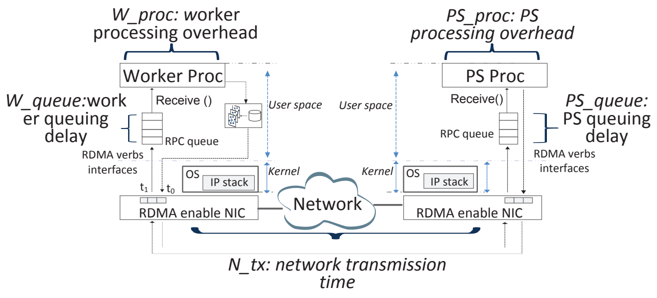


Figure 3: Five latency metrics in our data.

- *W_proc (worker side)* measures the time that the worker spent in processing the RPC response (*e.g.* parsing the model parameters pulled from the PS node.). Note that it does not contain the time for model training.
- *W_queue (worker side)* measures the wait time in the queue. This is because all received RPC responses are first pushed into a software receiving queue, waiting for the worker to read and parse them.
- *PS_proc (PS side)* measures the time that the worker spends in processing the RPC request. For a pull operation (*i.e.* requesting parameter values), this is to prepare the requested model parameters along with their values; for a push operation (*i.e.* updating parameters), this process includes an arithmetic calculation to aggregate gradient parameters, and a write operation to update the parameters with the values contained in the message. This metric along with the next (*PS_queue*) are piggybacked on the response to the worker, which finally outputs the log for this RPC.
- *PS_queue (PS side)* measures the wait time of the request message in its receiving queue (similar to *W_queue*).
- *N_tx* measures the transmission time over the network (including network stack processing time). The RPC message sent to the PS contains the timestamp (t_0) recorded at the worker when it is sent to the network stack. The PS copies t_0 to the response, which arrives at the worker's receiving queue at t_1. Then $N\_tx = t_1 - t_0 - PS\_queue - PS\_proc$.

Concurrent active connections. A PS node records the timestamp and worker ID for every RPC request. It divides the whole training process into windows of equal duration, t_{window}. For each window, it counts the unique worker IDs as the concurrent active connections.

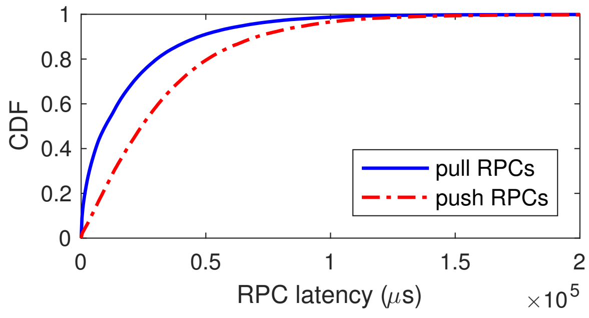
Parameter statistics . A PS node records for each parameter the number of gradients it received from the workers during the whole training process. After finishing the whole training process, the PS nodes dump its local statistics to a log server, which aggregates the statistics and generates for each parameter the total number of updates.

3 ANALYSIS OF DDL LATENCY

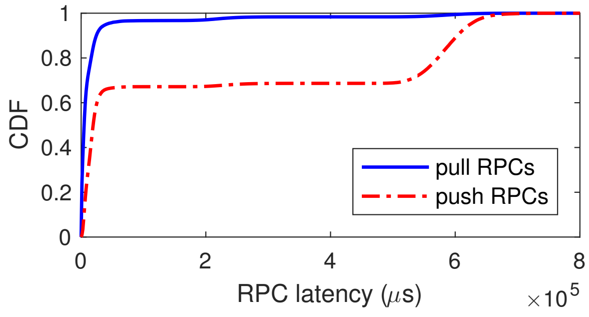
We next present our experimental results, characterizing the latency of DDL training.

3.1 Latency Overview and Breakdown

As already mentioned, training models involves *push* and *pull* operations which are, in essence, a series of RPC calls. In this section, we first give an overview of RPC latency for *push* and *pull* respectively, and then break down each RPC call into the five metrics.



(a) CDF of latency in R1.

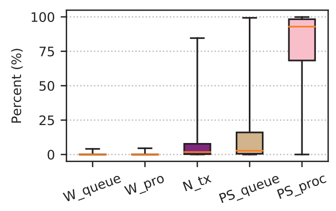


(b) CDF of latency in R2.

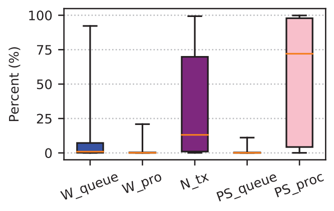
Figure 4: Distribution of latency in R1 and R2.

Distribution of Latency. We utilize the latency data of R1 and R2 to collect the total latency of each RPC calls. Figure 4 presents CDFs of the RPC latencies.[1] We observe a heavy tail distribution, in which the *push* operation is larger than that of the *pull* operation. This is because *push* operations enable workers to transmit their trained gradients to the centralized PS node, which not only involves large payloads but also creates an *in-cast* problem. We

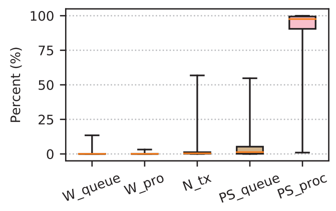
[1]Note that each RPC latency involves all five latency metrics described above.



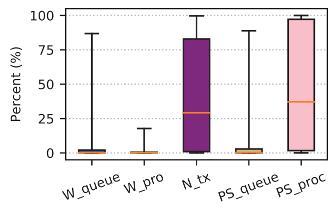
(a) *Push* operation in R_1.



(b) *Pull* operation in R_1.



(c) *Push* operation in R_2.



(d) *Pull* operation in R_2.

Figure 5: Each portion overhead of push and pull operations.

find that one *push* operation consists of one or more RPC requests for transmitting different layer gradients of the neural network. Consequently, the *push* operation ends only when the worker has received all corresponding RPC responses. Thus the cost of the *push* operation does depend on the last RPC processing. This decreases the performance. In addition, the measurement results in R2 (see Figure 4(b)) show that the *push* operations contain some centralized long latency, as indicated by the spike around 6x10^5. This is driven by the dependency between (push) RPC invocations at the PS, which means that one RPC may need to wait for the results of another RPC, which inevitably incurs long tail latency.

Breakdown. To explore the key components of the above RPC delays, we next quantify the cost of each portion in one RPC. Note that we do not change the breakdown over iterations. Figure 5 shows the breakdown of time spent on each metric for push and pull operations. This reveals that the primary performance costs involves two parts: the computation task processing in PS nodes (denoted by *PS_proc*) and the network transmission time (denoted by *N_tx*).

The proportions of the total RPC cost are different for *pull* and *push* operations though. Overall, the *PS_proc* proportion in *pull* operation is larger than that in *push*. This applies to both R1 and R2 latency data. On the contrary, *N_tx* holds a larger proportion in *pull* operations. A possible reason is that the *pull* operation makes the PS node transmit the up-to-date model parameters, each of which is represented by a 32-bit floating-point number, to workers. This leads to more communication volumes. Thus, we see that *PS_proc* and *N_tx* occupy the largest portion of RPC latency. The next section explores this in more detail.

3.2 Explaining Possible Bottlenecks

The previous section has shown that *PS_proc* and *N_tx* contribute the most to training delay. Overall, we find out three key causes by correlation analysis as follows.

RPC message size. As mentioned earlier, *N_tx* constitutes a larger portion of *pull* operations than *push* operations. We believe that the traffic volume is its key cause. Note that we collect only the size of RPC response messages from R1 latency data. This is because we find the size of RPC request messages is relatively fixed and small (only 100~ 200 bytes) due to message segmentation. Figure 6 presents a CDF of the response size, for both push and pull invocations. We see that the *pull* operation is much larger than the *push* operation. This is because the response messages of *push* operation often play a TCP ACK like role, and therefore its message size

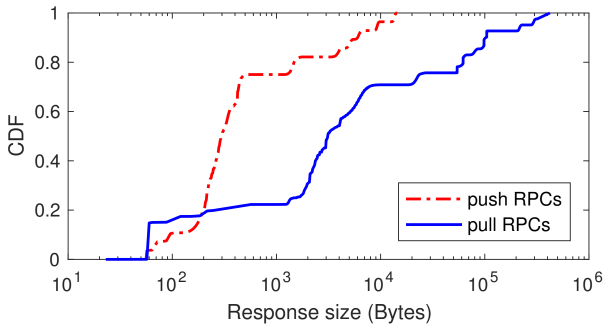
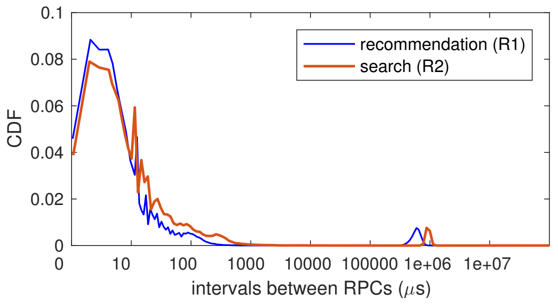


Figure 6: The response message size in *pull* and *push* operation.

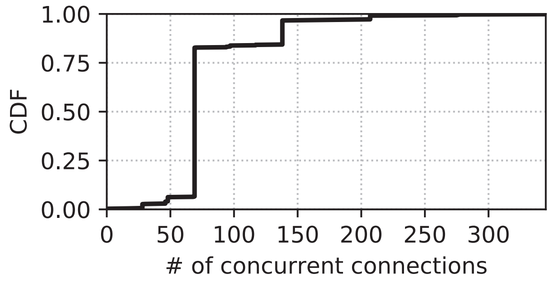
is limited. In a word, *N_tx* is positively correlated with the RPC message size.

Communication frequency. As previously identified, *PS_proc* occupies a large proportion of the total RPC cost. However, *PS_proc* only involves the read parameters in a *pull* operation, while the arithmetic calculation and the write parameters are included in a *push* operation. We assume that the overhead of read and write operations is identical. Consequently, the cost of arithmetic calculation can be estimated by the average overhead of *PS_proc* in push operations minus that in pull operations. This shows that the cost of an arithmetic computation is relatively small, occupying no more than 2.5% of the total *PS_proc* for a push operation. Consequently, the bottleneck of *PS_proc* may be the frequent (concurrent) read and write operations. In other words, frequent communication leads to a high *PS_proc* cost.

To confirm this, we explore the communication frequency between workers and PS nodes. To this end, we analyze the RPC time intervals between one PS node and one worker based on the R1 and R2 latency data. We find out that the interval of two consecutive RPCs is always very short while only a few intervals are long (see Figure 7(a)). This is because one *push* or *pull* operation will be divided into multiple consecutive RPCs so that their intervals are short. The long intervals are caused by the computation task (a.k.a training) on the workers. Consequently, in a short time interval, the PS node has to process multiple RPCs from one worker. In addition, we count the number of unique connections operated by the PS node in a very short interval (*i.e.* 1,000 μs). Figure 7(b) shows the result that the PS node processes 50~200 unique connections within 1,000 μs. Thus, it is clear that the communication frequency has an impact on the PS node.



(a) Frequency distribution for the time intervals in R2



(b) # of concurrent connections that are processed by one PS.

Figure 7: Distribution of communication time intervals between workers and the PS node (a); and the number of concurrent connections per PS within 1,000 μs (b).

Concurrent connections. Frequent concurrent communication may also lead to increases in *PS_proc*. To this end, we conduct two experiments as follows.

We first measure the CPU cost both on the worker and PS sides under six different setups, which we delineate in Table 2. Each setup varies the number of workers and PS nodes. Figure 8 presents the CPU utilization across each setup. As we increase the scale of the training system, the CPU utilization rate on the worker side is reduced while that of the PS side is increased. In other words, the PS node becomes more heavily loaded, and becomes a key bottleneck, achieving 90% utilization rate at most. The larger scale setup we use, the more concurrent active connections the PS node needs to manage because it needs to connect to all workers. Confirming intuition, this means that the resource consumption on the PS increases as the number of workers grows.

An obvious conjecture is that this is caused by the computational load placed on PS nodes. To control for this, we eliminate the effects of read-write operations by sending empty RPC requests to one PS node. Figure 9 shows the results. We see that, regardless of read-write operations, the delay suffers linear increases as the number of connections grow. This shows that concurrent active connections do lead to significant overhead. This is because they not only introduce concurrent read-write operations which lead to competition, but also incur connection costs, *e.g.* context switches, connection tracking and status maintenance.

4 TOWARDS BOTTLENECK MITIGATION

Offloading Computation. The above has shown one of the performance bottlenecks is concurrent read' and write operations in the PS node. Recently, programmable switches have been introduced and deployed in data center networking. Similarly, in-network computation has emerged [25] by exploiting the computation capability of the network. Thus, we can potentially offload the read/write operations (*e.g.* gradient aggregation) to the network in order to reduce the overhead of the PS node. Some recent works also discuss such explorations [21, 22].

Unfortunately, the memory of the programmable switches only contains tens of megabytes. Hence, they cannot store all parameters (*e.g.* 150 million parameters in R_1). If we randomly offload the operations of only some parameters (*e.g.* tens of thousands parameters), it will not bring significant improvements. Even if we adopt some cache mechanisms (*e.g.* Least Recently Used, LRU), it may lead to a large number of cache misses and lower down the performance.

Table 2: The different setups of the system. Note that the scale of worker nodes and PS nodes is preserved in a proportion of 4 to 1.

Setup	S1	S2	S3	S4	S5	S6
# workers	80	120	160	200	240	280
# PS nodes	20	30	40	50	60	70

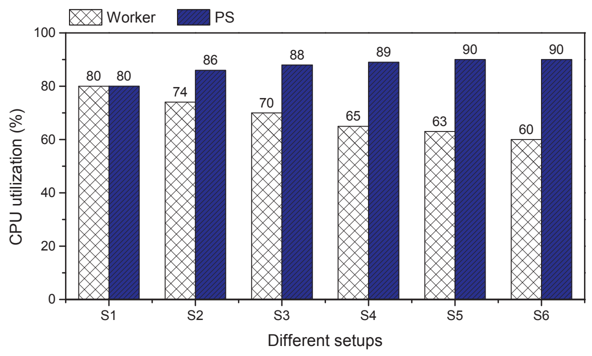


Figure 8: CPU utilisation when varying the scale of the distributed DL cluster.

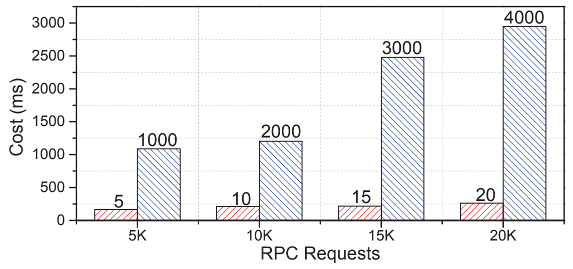


Figure 9: The cost of concurrent connections. This cost is introduced by connection management on the server side. We use different scales of connections to perform the same amount of RPC requests. The numbers on top of the histograms represent the volume of concurrent connections sending x RPC requests.

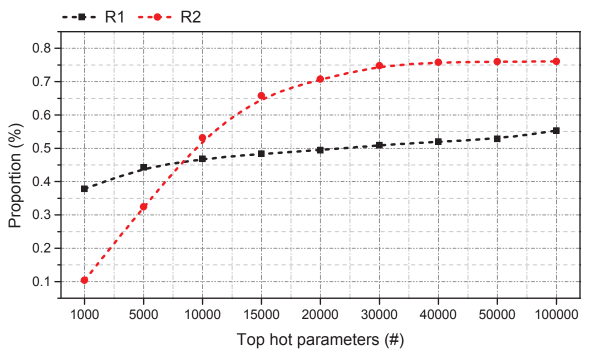


Figure 10: The distribution of the hot parameters in high-dimensional sparse application.

To explore this, we analyze our high-dimensional sparse datasets from Alibaba. Figure 10 presents the proportion of communication frequency for buckets of the top parameters. This shows that they contain "hot" parameters (*i.e.* those that are often read or written). Across the 150 million parameters, the top 20,000 parameters constitute over half of all updates. We can exploit this skewed distribution to only offload the read/write operations of the top hot parameters to the switches, and the rest of the parameters (a.k.a cold parameters) can be processed in the PS node side. If we offload the top 20,000 hot parameters, half of the total read/write operations will be offloaded to switches. Consequently, 50% of *PS_proc* can be saved.

Streamlining Concurrent Connections. As mentioned earlier, the number of concurrent connections is also one of the key factors that affects performance. Thus, reducing the number of connections to the PS node could improve performance. We could exploit the capability of the programmable switches and aggregate connections between workers and PS nodes. In essence, switches could be viewed as a connection proxy. Although modern programmable switches are often equipped with a general CPU, it is unlikely capable of running a connection proxy. We therefore argue that simplified network, transport or application protocols that can enable connection aggregation would be useful.

5 RELATED WORK

Distributed DL has emerged as a common practice, widely used in a range of areas [7, 13, 20]. However, its performance often suffers from distributed communication [26, 27]. To this end, there are numerous existing works for speeding up the communication in DDL training. NCCL [4], MPI [10] and Gloo [2] offer high-performance communication libraries to improve the communication efficiency. RDMA [16] is adopted to accelerate data transmission with an extra in-network support. Another direction that has been explored is compressing data transmission volume via gradient quantization or sparse parameter synchronization [24, 26].

Others have focused on network flow scheduling and communication scheduling. The former is to minimize the flow completion time by using flow control [8, 18]; the latter decouples the dependence between gradients and changes their transmission order to reduce communication overhead, such as ByteScheduler [19]. Recently, in-network computation has emerged [25], and some approaches propose to utilize programmable switches to improve the training efficiency [15, 21, 22].

Our work differs in that we are among the first to characterize the communication latency in a practical DDL system. Specially, we characterized the communication patterns and identify key performance bottlenecks. Our results can inform further system innovation.

6 DISCUSSION AND FURTHER WORK

Summary. DDL has been used in a range of applications. However, the introduction of distributed computation means that performance is impacted by the network. This paper has studied the communication latency of Alibaba's DDL system using two high-dimensional sparse data applications, relying on 1.38M RPCs logs. We have shown that the main bottlenecks of the communication involve concurrent write/read operations and the connection management. We further find a skewed distribution of update frequency of individual parameters, which motivates the use of in-network computation to offload operations and reduce concurrent connections. We believe that our work can pave the way to optimize the distributed DL training.

Generalization of results. Though our measurement study only focuses on Alibaba's DDL platform, the results are generalizable for other DDL platforms for two reasons: (*i*) Alibaba's DDL platform follows the popular parameter server (PS) architecture, which has been widely adopted in industry (*e.g.* Microsoft Multiverso [3]); (*ii*) We train two high-dimensional sparse models for common industrial applications (*i.e.* advertising recommendation and search engine) and measure their communication patterns on Alibaba's DDL platform. We believe this is enough to capture a broad range of communication features during DDL training.

Mitigation techniques. Overall, the measurement results show that large the number of concurrent active connections and frequent communications results in the PS becoming the bottleneck. Thus we discuss the potential of using in-network computation and propose two mitigation techniques: (*i*) aggregating active connections on programmable switches; (*ii*) offloading some read/write operations to the switches as well. For other potential techniques, we can consider optimizing the host stack, replacing RPC mechanism or aggregating small messages.

Fine-grained measurements. Our current measurement metrics mainly focus on processing latency. In addition, for the PS processing, our metrics are relatively coarse-grained given the complicated tasks the servers process. In our future work, we will consider other metrics, such as memory usage, to perform more fined-grained measurement on the PS processing under different system choices.

ACKNOWLEDGMENTS

We thank our shepherd Alan Mislove and the anonymous reviewers for their insightful feedback. This work is supported in part by National Key R&D Program of China (Grant No. 2019YFB1802800), the National Science Fund of China (Grant No. 61725206) and the Youth Innovation Promotion Association CAS. The corresponding author is Zhenyu Li.

REFERENCES

[1] [n. d.]. The Apache Software Foundation. Apache hadoop. http://hadoop.apache.org/core/.. ([n. d.]).

[2] [n. d.]. Facebook, Gloo. https://github.com/facebookincubator/gloo. ([n. d.]).

[3] [n. d.]. Microsoft multiverso. https://github.com/Microsoft/multiverso/wiki.. ([n. d.]).

[4] [n. d.]. NVIDIA Collective Communication Library (NCCL). https://developer.nvidia.com/nccl. ([n. d.]).

[5] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2017. QSGD: Communication-efficient SGD via gradient quantization and encoding. In *Advances in Neural Information Processing Systems*. 1709–1720.

[6] Ashish Goel Bahman Bahmani and Rajendra Shinde. 2012. Efficient distributed locality sensitive hashing. In *ACM CIKM*.

[7] Kyunghyun Cho, Bart Van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *arXiv: Computation and Language* (2014).

[8] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. 2014. Efficient coflow scheduling with varys. In *Proceedings of the 2014 ACM conference on SIGCOMM*. 443–454.

[9] Zhong Deng Gaurav Soni Jianxi Ye Jitu Padhye Chuanxiong Guo, HaitaoWu and Marina Lipshteyn. 2016. RDMA over Commodity Ethernet at Scale. In *ACM SIGCOMM*.

[10] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. 2004. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 97–104.

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

[12] Biye Jiang, Chao Deng, Huimin Yi, Zelin Hu, Guorui Zhou, Yang Zheng, Sui Huang, Xinyang Guo, Dongyue Wang, Yue Song, et al. 2019. XDL: an industrial deep learning framework for high-dimensional sparse data. In *Proceedings of the 1st International Workshop on Deep Learning Practice for High-Dimensional Sparse Data*. 1–9.

[13] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. (2012), 1097–1105.

[14] 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 *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 583–598.

[15] Youjie Li, Iou-Jen Liu, Yifan Yuan, Deming Chen, Alexander Schwing, and Jian Huang. 2019. Accelerating distributed reinforcement learning with in-switch computing. In *Proceedings of the 46th International Symposium on Computer Architecture*. 279–291.

[16] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K Panda. 2004. High performance RDMA-based MPI implementation over InfiniBand. *International Journal of Parallel Programming* 32, 3 (2004), 167–198.

[17] T. Das A. Dave J. M. Ma M. McCauley M. J. Franklin S. Shenker M. Zaharia, M. Chowdhury and I. Stoica. 2012. Fast and interactive analytics over Hadoop data with Spark. In *USENIX ;login:*.

[18] Luo Mai, Chuntao Hong, and Paolo Costa. 2015. Optimizing network performance in distributed machine learning. In *7th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 15)*.

[19] 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 *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 16–29.

[20] Hasim Sak, Andrew W Senior, and Francoise Beaufays. 2014. Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling. (2014), 338–342.

[21] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. 2017. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*. 150–156.

[22] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. 2019. Scaling distributed machine learning with in-network aggregation. *arXiv preprint arXiv:1903.06701* (2019).

[23] Sangeetha Abdu Jyothi Sayed Hadi Hashemi and Roy H. Campbell. 2019. TicTac: Accelerating distributed deep learning with communication scheduling. *SysML* (2019).

[24] Ananda Theertha Suresh, Felix X Yu, Sanjiv Kumar, and H Brendan McMahan. 2017. Distributed mean estimation with limited communication. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 3329–3337.

[25] Yuta Tokusashi, Huynh Tu Dang, Fernando Pedone, Robert Soulé, and Noa Zilberman. 2019. The case for in-network computing on demand. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16.

[26] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2017. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Advances in neural information processing systems*. 1509–1519.

[27] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing. 2017. Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 181–193.