Improving Network Performance for Distributed Machine Learning in Commodity Clusters with Parameter Flow

Future Network Theory Lab, Huawei Research

Abstract—The Parameter Server (PS) framework is widely used to train machine learning (ML) models in parallel. It tackles the big data problem by having *worker* nodes performing data-parallel computation, and having *server* nodes maintaining globally shared *parameters*. When training big models, *worker* nodes frequently pull *parameters* from *server* nodes and push *updates* to *server* nodes, resulting in high communication overhead. Our investigations showed that modern distributed ML applications could spend up to 5.7 times more time on communication than computation. To address this problem, we propose a novel communication layer for the PS framework named Parameter Flow (PF) with three techniques. First, we introduce an update-centric communication (UCC) model to exchange data between *worker/server* nodes via two operations: broadcast and push. Second, we develop a dynamic value-bounded filter (DVF) to reduce network traffic by selectively dropping *updates* before transmission. Third, we design a tree-based streaming broadcasting (TSB) system to efficiently broadcast aggregated *updates* among *worker* nodes. PF could significantly reduce network traffic and communication time. Extensive performance evaluations showed that PF could speed up popular distributed ML applications by a factor of up to 4.3 in a dedicated cluster, and up to 8.2 in a shared cluster, compared to a generic PS system without PF.

Index Terms—Machine Learning, Distributed System, Communication Overhead, Parameter Server

1 Introduction

Machine Learning (ML) builds models from training data, and use them to make predictions on new data. It is used in a wide range of applications, including image recognition, natural language processing and recommender systems [1], [2]. Typically, an ML model consists of a large number of *parameters*, represented as vectors and matrices. To minimize the prediction error, an ML application usually uses an iterative-convergent algorithm, such as stochastic gradient descent (SGD), to train a given ML model.

In industrial ML applications, the size of training data sets could be hundreds to thousands of terabytes. For example, the Yahoo News Feed dataset stands at roughly 110 billion lines of user-news interaction data; ImageNet contains approximately 14 million labeled images; ClueWeb12 has 733 million web pages. Due to the limitation in storage and computation resource, modern single-node ML systems like Theano cannot cope with such large training datasets.

To handle big data, various distributed ML systems have been proposed based on the Parameter Server (PS) framework [3], such as Petuum [4], MxNet [5], Project Adam [6], SINGA [7], and TensorFlow [8]. The PS framework can scale to large cluster deployments by having *worker* nodes performing data-parallel computation, and having *server* nodes maintaining globally shared *parameters*. When training ML models, *worker* nodes continuously pull latest *parameters* from *server* nodes, perform computation on partitions of the training dataset, and push generated *updates* to *server* nodes. The *server* nodes would aggregate all received *updates*, and return a new version of *parameters*. This is done iteratively to bring *parameters* closer to the optimal value.

A major challenge of distributed ML is the high communication overhead. Modern ML applications are trending to learn big models with hundreds of millions of *parameters*. For example, VGG-19 [9] contains 175M *parameters*, and the multi-class logistic regression (MLR) model trained from Wikipedia consists of billions of *parameters* [10]. When using the PS framework, *worker* nodes should push all *updates* and pull a huge amount of *parameters* frequently, generating a large amount of network traffic. Due to the limited bandwidth of commodity cluster network (i.e. 1 Gbps and 10 Gbps Ethernet), each *worker* node spends a significant portion of time on communication. Our experiments show that the communication time could be much more than the computation time in many cases. Hence, it is important to reduce the communication cost when learning big models with the PS framework.

Many approaches have been proposed to improve the network performance of distributed ML. For example, MPI-based systems, such as S-Caffe [11], Malt [12] and COTS-HPC [13], use InfiniBand [14] to transmit all *parameters* or *updates* with high bandwidth and low latency, and leverage CUDA-Aware techniques [11], [15], [16] to improve the performance of GPU-GPU communication in HPC clusters. Bounded Asynchronous Parallel (BAP) models [17] could reduce the frequency of push/pull operations, and overlap communication with computation. Li-PS [18] can use user-defined filters to select "important" *parameters* and *updates* for transmission. MLNet [19] builds an overlay network to aggregate *updates* on *worker* nodes. *Server* nodes only receive messages from a part of *worker* nodes to avoid congestion.

However, existing approaches have several limitations. First, in many cloud computing platforms, data centers and clusters, only commodity network is available, since high performance network fabrics like InfiniBand could increase infrastructure costs significantly [20] [19]. For example, our experiments showed that the network bandwidth between two EC2 instances ranges from 1Gbps to 10 Gbps. Due to the restricted network bandwidth, MPI-based approaches still have high communication overhead in commodity clusters. Second, the primary goal of BAP models is to reduce *worker* nodes' waiting delay in a PS-based distributed ML system: they do not directly address the network traffic and communication time problem. Third, existing filtering techniques are designed and optimized for some particular ML models and applications. For example, Li-PS could only filter *parameters* and *updates* represented by sparse vectors or matrices. Finally, MLNet only considers the push operation.

In this paper, we propose Parameter Flow (PF) to tackle the high communication overhead of distributed ML in commodity clusters. PF is a communication mechanism for the PS framework with three techniques. First, we design an update-centric communication (UCC) model to exchange data between *worker* and *server* nodes using two functions: broadcast and push. Specifically, each *worker* node pushes local *updates* to *server* nodes, and *server* nodes broadcast aggregated *updates* to all *worker* nodes. Compared to conventional push-pull mechanism, our proposed push-broadcast approach only transmits *updates* via network. Second, we develop a dynamic value-bounded filter (DVF). DVF allows *worker* and *server* nodes to selectively drop *updates* during the push and broadcast operations to reduce network traffic. Meanwhile, to ensure convergence, DVF guarantees a bound on the magnitude of dropped *updates*. Third, we design a tree-based streaming broadcasting (TSB) system to further reduce the communication time when propagating aggregated *updates* from *server* nodes to *worker* nodes. As a result, PF could significantly reduce the communication overhead of distributed ML in commodity clusters.

The primary contributions of this paper are as follows:

- We propose a push-broadcast communication mechanism named PF for the PS framework to improve the network performance of distributed ML in commodity clusters with UCC, DVF and TSB;
- We perform a comprehensive theoretical analysis on PF's convergence guarantees;
- We develop a working implementation of PF using ZMQ, and integrate it with popular ML engines to support a broad range of ML algorithms and models;
- We conduct extensive experiments using a testbed. The results show that PF could speed up distributed ML applications by a factor of up to 4.3 in a dedicated cluster, and up to 8.2 in a shared cluster, compared to a generic PS system using push-pull mechanism.

The rest of the paper is structured as follows. Section 2 presents backgrounds on ML, distributed ML, and the PS framework with push-pull communication mechanism. Section 3 introduces PF and its three techniques: UCC, DVF and TSB. Section 4 provides a theoretical analysis of PF. The evaluation results are detailed in Section 5. Section 6 summarizes related works. Section 7 concludes the paper.

TABLE 1
Summary of notations used.

Clock	an integer representing the training progress of a *worker node*, measured by the number of completed iterations.
$J(\cdot)$,	objective function of Problem **P1**
$f(\cdot)$	loss function of Problem **P1**
$r(\cdot)$	regularizer function of Problem **P1**
$\Delta(\cdot)$	function to compute *updates*
$F(\cdot)$	function to update *parameters* using received *updates*
$\boldsymbol{W}$	vector of *parameters* of an ML application
$\boldsymbol{U}$	vector of *updates* of an ML application
d	size of $\boldsymbol{W}$ and $\boldsymbol{U}$
η	initial learning rate of of an ML algorithm
$\mathcal{D}$	training dataset of an ML application
m	number of examples of training dataset
p	number of *server* nodes of a PS system
n	number of *worker* nodes of a PS system
σ	initial bounded value of dropped *updates* in DVF
δ	initial threshold used to selectively drop *updates* in DVF
T	maximum degree of the tree used in TSB
H	maximum depth of the tree used in TSB
$c_{i,j}$	communication cost from node i to node j in TSB
$l_{i,j}$	$\in \{0,1\}$, 1 denotes node i sends data to node j in TSB
$h^k_{i,j}$	$\in \{0,1\}$, 1 denotes the path from the root node to node k contains the connection from node i to node j in TSB
t_i	$\in \{0,1\}$, 1 denotes node i is a leaf node in TSB

2 Distributed ML on the PS Framework

In this section, we summarize basic concepts on ML. Then, we describe data-parallel ML algorithms on the PS framework. Finally, we identify the communication overhead of PS-based ML applications via experiments. Table 1 shows the notations and their definitions used in the paper.

2.1 ML: A Primer

The goal of ML is to learn models from the training dataset, and use them to make predictions on new data. Typically, an ML model consists of a large number of *parameters*. In this paper, we use a vector $\boldsymbol{W}$ to represent all *parameters* of an ML model. An ML application could be described as an optimization problem: given training dataset $\mathcal{D}$ with m examples, it tries to find optimum $\boldsymbol{W}$ to minimize the objective function $J(\cdot)$,

$$\textbf{P1:} \quad \min_{\boldsymbol{W}} \quad J(\boldsymbol{W}) = \sum_{i=1}^{m} f(\boldsymbol{W}, \mathcal{D}_i) + r(\boldsymbol{W}), \qquad (1)$$

where $f(\cdot)$ is the *loss function*, such as $\mathcal{L}_2$ loss and quadratic loss, to represent the prediction error on one example of training dataset, and $r(\cdot)$ is the regularizer function, such as $\mathcal{L}_1$ regularization and $\mathcal{L}_2$ regularization, to limit ML models' complexity. **P1** could be used to represent a wide range of ML models, such as logistic regression (LR), matrix factorization (MF) and convolutional neural networks (CNNs).

It is common to use an iterative-convergent algorithm to solve **P1**, for example gradient descent (GD) and stochastic gradient descent (SGD). An iterative-convergent ML algorithm usually executes following equation iteratively until some convergence criteria are met:

$$\boldsymbol{U}^t = \Delta(\boldsymbol{W}^t, \mathcal{D}), \text{ then } \boldsymbol{W}^{t+1} = F(\boldsymbol{W}^t, \boldsymbol{U}^t), \qquad (2)$$

2.2 Distributed ML with the Push-Pull Mechanism

To cope with big training data sets, many distributed ML systems, such as Petuum, MxNet, Adam, SINGA and TensorFlow, have been proposed. These systems are designed

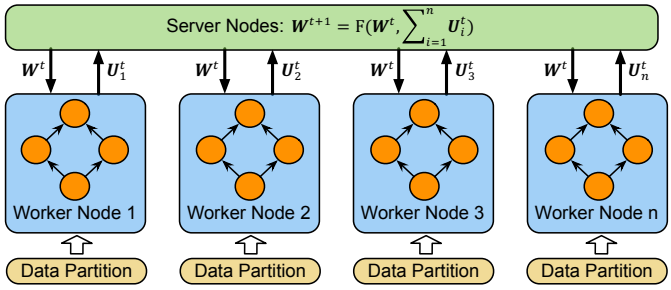


Fig. 1. System architecture of the PS framework. In PS, *worker* nodes are in charge of computing *updates*, and *server* nodes manage globally shared *parameters*. PS uses a push-pull communication mechanism to exchange data between *worker* nodes and *server* nodes.

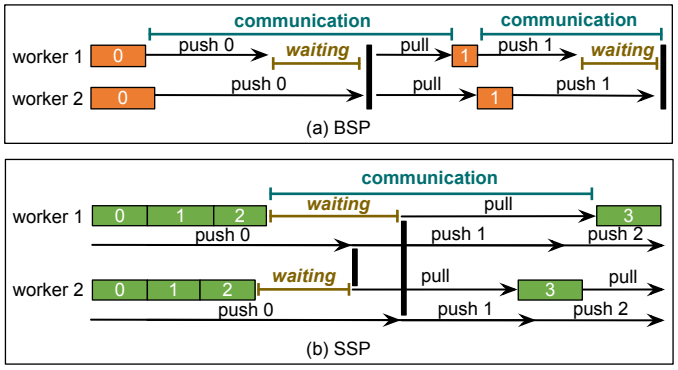


Fig. 2. The BSP and SSP syhchronization models. With BSP, there is a synchronization barrier at the end of each iteration. With SSP, *worker* nodes can use cached *parameters* to perform computation. In this example, the staleness threshold is 2. Thus, to perform computation of clock 3, a *worker* node needs to see effects of *updates* with timestamp 0 from all other *worker* node.

based on the PS framework, and execute data-parallel ML algorithms[1] [4] with pull and push operations. As shown in Figure 1, the PS framework contains a group of *server* nodes and a group of *worker* nodes. *Parameters* are globally shared and managed on *server* nodes. Training dataset is partitioned and assigned to *worker* nodes.

A data-parallel ML algorithm usually executes following equation iteratively on the PS framework with push-pull operations until some convergence criteria are met:

$$\begin{aligned} &\text{Worker Nodes}: \quad \boldsymbol{U}_i^t = \Delta(\boldsymbol{W}^t, \mathcal{D}_i), \\ &\text{Server Nodes}: \quad \boldsymbol{W}^{t+1} = F(\boldsymbol{W}^t, \textstyle\sum_{i=1}^{n} \boldsymbol{U}_i^t), \end{aligned} \tag{3}$$

where i is the index of the i-th *worker* node. Progress of a *worker* node is represent by "clock", which is measured by the number of completed iterations. *Updates* generated at clock t are timestamped with t. During the training process, a *worker* node continuously performs computation on $\boldsymbol{W}$ and outputs update vector $\boldsymbol{U}$, which is aggregated on *server* nodes to update $\boldsymbol{W}$. To exchange data between *worker* nodes and *server* nodes, the PS framework defines a push-pull communication model: *worker* nodes pull latest $\boldsymbol{W}$ from *server* nodes, and push newest $\boldsymbol{U}$ to *server* nodes.

1. Model-parallelism is another approach, which is designed to deal with ML applications with extreme-big models. In this paper, we only consider data-parallelism computation model.

2.3 Synchronization Models

Two types of synchronization models are widely used on the PS framework when executing data-parallel ML algorithms: Bulk Synchronous Parallel (BSP) and Bounded Asynchronous Parallel (BAP) [17].

2.3.1 BSP

BSP places a synchronization barrier at the end of each iteration when executing data-parallel ML algorithms. At clock t, every *worker* node needs to pull $\boldsymbol{W}^t$ and push $\boldsymbol{U}^t$. *Server* nodes update $\boldsymbol{W}^t$ to $\boldsymbol{W}^{t+1}$ after receiving $\boldsymbol{U}^t$ from all *worker* nodes. With BSP, distributed ML applications may have high waiting delay, since fast *worker* nodes must wait for the slow ones at every iteration, as shown in Figure 2(a).

2.3.2 BAP

BAP relaxes the consistency guarantees to reduce waiting delay, and bounds the amount of inconsistency to ensure convergence [17]. Since there is no synchronization barrier, *worker* nodes can have different clocks. Several BAP models have been proposed:

- **Stale Synchronous Parallel (SSP)** [17]. SSP guarantees a bound on the clock difference between the fastest and slowest *worker* node. It allows a *worker* node with clock t to use cached *parameters*, if the cached *parameters* can see effects of all other nodes' *updates* from clock 0 to $t - s - 1$, where s is the staleness threshold. In this paper, we adopt the latest approach from [21], [22], in which SSP could overlap communication with computation.
- **Value-bounded Synchronous Parallel (VAP)** [21]. VAP guarantees a bound on the magnitude of the sum of *in-transit updates* from all *worker* nodes. An update is *in-transit* if it has not been by every *worker* node. VAP allows a *worker* node to perform computation without waiting for other ones, if the magnitude of the sum of all *in-transit updates* is less than v, where v is a user-defined threshold. However, it is difficult to implement VAP due to its strong condition [21].

Due to the relaxed consistency guarantee, SSP and VAP can reduce the communication cost of distributed ML to a certain extent. First, these two models reduce the waiting delay, since fast *worker* nodes would not always wait for the slow ones. Second, SSP and VAP could reduce the frequency of push/pull operations and overlap communication with computation [21].

Nevertheless, network communication is still a serious performance bottleneck for PS-based ML applications when learning big models with millions or billions of *parameters*. The primary goal of BAP models is to reduce *worker* nodes' waiting delay: they do not directly reduce the network traffic as well as the communication time. Thus, when pulling *parameters* or pushing *updates* via commodity networks, PS-based ML applications still have prolonged communication time even with BAP models, as shown in Figure 2(b).

2.4 Communication Overhead Analysis using AlexNet

To verify the high network communication overhead, we use PS-lite [3], which is an implementation of the PS framework and a core module of MxNet, to train AlexNet [23] on

ImageNet dataset with both BSP and SSP. In the experiment, we use five *worker* nodes and five *server* nodes connected via 1Gbps Ethernet. Each *worker* node has a NVIDIA K40 GPU. The staleness threshold of SSP is set to 4.

The results show that the communication overhead is quite significant with both BSP and SSP. More specifically, with BSP, each *worker* node needs to pull and push 61.3 million *parameters* and *updates* per iteration. As a result, the five *server* nodes need to receive and send out around 1,226MB data at each iteration. According to our measurement, in an iteration, the communication operation, including pull and push, would take around 6.9s on average. Compared to the computation time, which is around 1.2s per iteration, communication is the dominant component. With SSP, communication consumes around 3.6 times more time than computation through the overall training process. Compared to BSP, the performance gain of SSP comes from the lower waiting delay. However, due to the large network traffic and limited bandwidth, the training job still spends a significant portion of time on communication.

3 Parameter Flow Design

In this section, we describe the design of ParameterFlow (PF), and show how it can reduce communication overhead for distributed ML applications on the PS framework. We start with an overview of PF's architecture. Then, we introduce PF's three components: UCC, DVF and TSB.

3.1 System Overview

As shown in Figure 3, a PF-enabled PS framework has a similar system architecture to the conventional PS framework. It contains a group of *server* nodes and a group of *worker* nodes. Given a distributed ML application, its globally shared *parameters* are evenly partitioned and assigned to *server* nodes. The training dataset is partitioned and assigned to *worker* nodes. During the training process, a *worker* node performs data-parallel computation on *parameters* and its assigned training dataset partition, and pushes generated *updates* to *server* nodes, which aggregate them and produce a new version of *parameters*.

PF is a communication layer for the PS framework, running as a thread on *worker* nodes and *server* nodes. To reduce communication overhead, PF employs three techniques:

1) An update-centric communication (UCC) model. UCC exchanges data between *worker/server* nodes via two functions: broadcast and push. Specifically, each *worker* node pushes computed *updates* to *server* nodes, and *server* nodes broadcast aggregated *updates* to all *worker* nodes. *Worker* nodes would update *parameters* locally based on received aggregated *updates*. In this way, UCC only transmits *updates* over network.
2) A dynamic value-bounded filter (DVF). DVF allows *worker* nodes and *server* nodes to selectively drop *updates* during the push and broadcast operations to directly reduce network traffic.
3) A tree-based streaming broadcasting (TSB) system. TSB leverages an optimized tree-based overlay network to reduce communication time when broadcasting aggregated *updates* from *server* nodes to *worker* nodes.

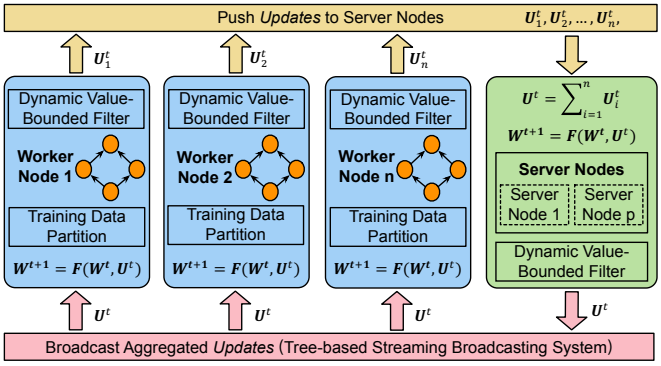


Fig. 3. System architecture of a PF-enabled PS framework with three key components: UCC, DVF and TSB.

3.2 Update-Centric Communication Model

PF transmits *updates* with broadcast and push operations. The data-parallel ML algorithm executes following equations iteratively until some convergence criteria are met:

$$\begin{aligned} &\text{Worker Nodes :} && \boldsymbol{U}_i^t = \Delta(\boldsymbol{W}^t, \mathcal{D}_i), \\ &\text{Server Nodes :} && \boldsymbol{U}^t = \sum_{i=1}^{n} \boldsymbol{U}_i^t, \\ &\text{Both Node Types :} && \boldsymbol{W}^{t+1} = F(\boldsymbol{W}^t, \boldsymbol{U}^t). \end{aligned} \tag{4}$$

At clock t, i-th *worker* node performs computation on its local set of *parameters* $\boldsymbol{W}^t$ and its assigned training dataset partition $\mathcal{D}_i$ to generate *updates* $\boldsymbol{U}_i^t$, then push $\boldsymbol{U}_i^t$ to *server* nodes as in the conventional PS framework. *Server* nodes aggregate received *updates* to a vector $\boldsymbol{U^t}$, and update $\boldsymbol{W}^t$ to $\boldsymbol{W}^{t+1}$ using function $F(\cdot)$. Instead of letting *worker* nodes pull $\boldsymbol{W}^{t+1}$ at clock $t+1$, PF broadcasts $\boldsymbol{U^t}$ to all *worker* nodes, which would update their local $\boldsymbol{W}^t$ to $\boldsymbol{W}^{t+1}$ based on the function $F(\cdot)$.

3.2.1 Synchronization Models

UCC can work in conjunction with BSP and BAP. We use SSP as an example of BAP due to its popularity.

UCC-BSP. *Server* nodes broadcast aggregated *updates* when receiving *updates* with the same timestamp from all *worker* nodes. After computing its new version of *parameters* based on the newly received aggregated *updates*, a worker node could perform the computation for the next iteration.

UCC-SSP. *Worker* nodes can use cached stale *parameters* within a staleness threshold to compute *updates*. This model also guarantees a bound on the clock difference between fastest and slowest *worker* nodes. It executes following equation iteratively until some convergence criteria are met:

$$\begin{aligned} &\text{Worker Nodes :} && \boldsymbol{U}_i^t = \Delta(\boldsymbol{W}^{\hat{t}}, \mathcal{D}_i), \text{ if } t - \hat{t} \leq s \\ &\text{Server Nodes :} && \boldsymbol{U}^t = \sum\nolimits_{i \in \mathcal{A}} \boldsymbol{U}_i^t + \sum\nolimits_{i \in \mathcal{B}} \sum\nolimits_{\bar{t}=t+1}^{t+s} \boldsymbol{U}_i^{\bar{t}}, \\ &\text{Both Node Types :} && \boldsymbol{W}^{t+1} = F(\boldsymbol{W}^t, \boldsymbol{U}^t), \end{aligned} \tag{5}$$

where s is the staleness threshold; $\mathcal{A}$ is the set of *worker* nodes which push "guaranteed" *updates* with timestamp t; $\mathcal{B}$ is the set of *worker* nodes which push "extra" *updates* with timestamp range $[t+1, t+s]$, and $|\mathcal{A}| + |\mathcal{B}| = n$. *Worker* nodes in UCC-SSP may be having different clocks, and may generate *updates* with different timestamps. *Server* nodes aggregate all received *updates* regardless of their timestamps.

When all *worker* nodes have pushed *updates* with timestamp t, the *server* nodes finalize $\boldsymbol{U}^t$, and broadcast it. As a result, $\boldsymbol{W}^{t+1}$ could reflect the changes of two part of *updates*: i) "guaranteed"*updates* with the timestamp range of $[0, t]$ from all *worker* nodes; and ii) "extra" *updates* with a timestamp range of $[t+1, t+s]$ from some fast *worker* nodes. The *worker* node would be blocked when its clock minus its parameter version number is larger than the staleness threshold.

3.2.2 UCC Analysis

Compared to the conventional push-pull communication mechanism, which transmits both *parameters* and *updates*, UCC makes PF transmit only *updates* via the network using push and broadcast operations. Though UCC would not directly improve system performance, it provides opportunities to reduce communication overhead, since *updates* have more opportunities for reducing communication overhead. For example, when training matrix-parameterized models, a low-rank matrix $\boldsymbol{U}$ contains all *updates* [21]. Thus, $\boldsymbol{U}$ can be represented by two vectors (i.e. $\boldsymbol{U} = \boldsymbol{u}\boldsymbol{v}^{\mathrm{T}}$), which would significantly reduce the amount of data to be transmitted. In contrast, if $\boldsymbol{U}$ is accumulated into *parameters*, the low-rank property of $\boldsymbol{U}$ would be lost.

3.3 Dynamic Value-Bounded Filter

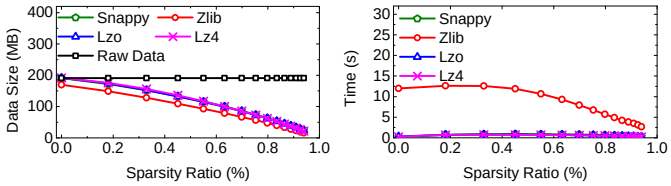
DVF allows *worker* nodes and *server* nodes to drop *updates* selectively during the push and broadcast operations, while guaranteeing that the magnitude of dropped *updates* is less than a bounded value. Given an update vector $\boldsymbol{U}$, an *update* is dropped if the corresponding entry in $\boldsymbol{U}$ is set to 0 by DVF. Thus, PF would not transmit all *updates*: $\boldsymbol{U}$ is transformed into a sparse vector, which can be compressed significantly for network transmission. In this way, *worker* nodes and *server* nodes can reduce network traffic during the push and broadcast operations.

3.3.1 Motivation

We design DVF to use a sparse vector to represent *updates* in transmission for two reasons: i) sparse vectors could have a good compression ratio[2] and acceptable compression time; and ii) data-parallel ML algorithms could converge well in this case under some particular conditions.

High Compression Ratio. We use experiments to illustrate the compression ratio and compression time of sparse vectors. In the experiments, we use 4 libraries (*snappy*, *zlib*, *lzo* and *lz4*) to compress a vector with various sparsity ratio[3] on an Intel Core i5 (2.4 GHz). More specifically, we adopt the hybrid approach shown in [24], [25]. If the sparsity ratio is higher than a given threshold (in this paper, this threshold is set to 0.8), we convert the input vector into a non-zero array and a index array. Otherwise, we store all values in a single array. Then, compression libraries convert the input vector into a string. Figure 4(a) shows that all 4 libraries generate a string with much smaller data size than the raw data, which stores all values in a single array without compression. For example, *snappy* could reduce data size by a factor of 3.1 for a vector of size 50M with

2. Data compression ratio = raw data size/compressed data size.
3. For vector $v = \mathbb{R}^d$, sparsity ratio $= 1 - ||v||_0/d$.



(a) Traffic Size (b) Compression Time

Fig. 4. Data size and compression time comparison between 4 compression libraries. The vector contains 50M Float32 numbers.

80% sparsity ratio. Though *snappy*, *lzo* and *lz4* have a little lower data compression ratio than *zlib*, they achieve much faster compression speed. For example, Figure 4(b) shows that *snappy*, *lzo* and *lz4* consume roughly 0.62s, 0.51s and 0.49s to compress a vector of size 50M with 80% sparsity ratio, respectively. The corresponding value for *zlib* is 5.71s. It should be noted that we use a single thread to perform all experiments. When using multi-threads, we could have higher compression speed. In this work, PF uses snappy to balance compression ratio and compressing time.

Convergence Guarantee. Data-parallel ML algorithms can converge if we guarantee a bound on the magnitude of dropped *updates*. DVF allows *server* nodes and *worker* nodes to drop *updates* selectively. Thus, *worker* nodes may perform computation for the next iteration without receiving all *updates*. As discussed in [21], [22], data-parallel ML algorithms can converge as long as the magnitude of the sum of *in-transit* update vectors, which have not been seen by all *worker* nodes, is less than a given threshold. Using this property, we set a bound on the magnitude of dropped *updates* during the push and broadcast operations, and find that this strategy could guarantee convergence.

3.3.2 Dropping Updates Selectively

When pushing or broadcasting an update vector $\boldsymbol{U}^t$, DVF selectively drops some entries of $\boldsymbol{U}^t$. Let $\boldsymbol{U}^t_{drop}$ be the vector containing all dropped entries, which is managed by the local storage of *worker* nodes or *server* nodes. Let $\boldsymbol{U}^t_{rmn}$ be the vector with remaining entries, which would be pushed or broadcast after compression. It should be noted that DVF does not remove $\boldsymbol{U}^t_{drop}$. In the next communication operation, $\boldsymbol{U}^t_{drop}$ would be accumulated into the newly generated update vector as

$$\boldsymbol{U}^{t+1} \leftarrow \boldsymbol{U}^t_{drop} + \boldsymbol{U}^{t+1}. \tag{6}$$

In this way, DVF would not lose any update information during the training process, so convergence guarantee would not be affected.

DVF needs to find $\boldsymbol{U}^t_{rmn}$ with the maximum sparsity ratio (i.e. minimum $||\boldsymbol{U}^t_{rmn}||_0$) for high compression ratio, while making sure that the magnitude of $\boldsymbol{U}^t_{drop}$ is no greater than a bounded value σ^t. It can be formalized as a quadratically constrained program as follows:

$$\textbf{P2:} \quad \min \quad ||\boldsymbol{U}^t_{rmn}||_0, \tag{7}$$

$$\text{s.t.} \quad \boldsymbol{U}^t = \boldsymbol{U}^t_{rmn} + \boldsymbol{U}^t_{drop}, \tag{8}$$

$$||\boldsymbol{U}^t_{drop}||_2 \leq \sigma^t. \tag{9}$$

P2 is a NP-hard problem. To see this, we can transform **P2** into a binary integer programming problem as follows:

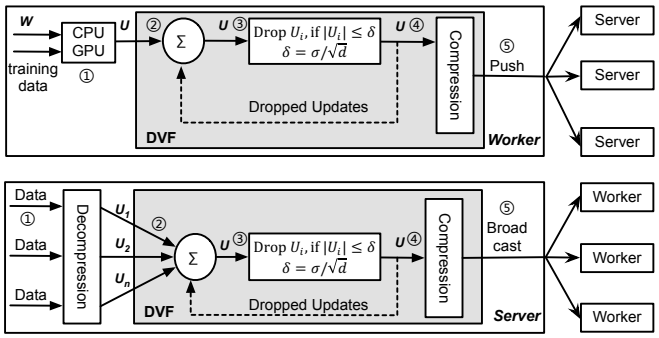


Fig. 5. DVF allows *worker* nodes and *server* nodes to selectively drop *updates* with a given threshold during the push or broadcast operations.

$\min\ ||\boldsymbol{U}^t \times \boldsymbol{r}||_0$, subject to $||\boldsymbol{U}^t - \boldsymbol{U}^t \times \boldsymbol{r}||_2 \le \sigma^t$, where $\boldsymbol{r}$ is a vector of binary variables. If $\boldsymbol{r}_i = 0$, $\boldsymbol{U}_i^t$ is dropped; otherwise, $\boldsymbol{U}_i^t$ is kept for transmission.

We simplifies **P2** by replacing its second constraint with $\sqrt{d}||\boldsymbol{U}_{drop}^t||_\infty \le \sigma^t$, since $||\boldsymbol{U}_{drop}^t||_2 \le \sqrt{d}||\boldsymbol{U}_{drop}^t||_\infty$. As a result, the simplified problem can be solved by a $\mathcal{O}(1)$ algorithm. More specifically, to minimize the objective function of **P2**, any *update* whose absolute value is no greater than the threshold $\delta^t = \sigma^t/\sqrt{d}$ should be dropped; and other *updates* would be kept for transmission.

As shown in Figure 5, DVF works on both *worker* nodes and *server* nodes to reduce network traffic with δ^t. Specifically, before the push and broadcast operations, DVF selectively drop some *updates* as follows:

$$\begin{cases} \boldsymbol{U}_{drop,i}^t = \boldsymbol{U}_i^t, \text{ and } \boldsymbol{U}_{rmn,i}^t = 0, & \text{if } |\boldsymbol{U}_i^t| \le \delta^t, \\ \boldsymbol{U}_{rmn,i}^t = \boldsymbol{U}_i^t, \text{ and } \boldsymbol{U}_{drop,i}^t = 0, & \text{if } |\boldsymbol{U}_i^t| > \delta^t, \end{cases} \delta^t = \frac{\sigma^t}{\sqrt{d}}. \tag{10}$$

Only $\boldsymbol{U}_{rmn}^t$ would be pushed or broadcast. In this way, DVF improves the sparsity ratio of the update vector in transmission, and bounds the magnitude of dropped *updates* vector on each node. In Section 4, we will show the convergence guarantee for DVF.

3.3.3 Dynamic Threshold

In DVF, the threshold δ^t are dynamic values based on the training progress of the data-parallel ML algorithms used and real-time network performance.

Time-Varying. δ^t is time-varying as follows:

$$\sigma^t = \frac{\sigma}{\sqrt{t}}, \text{ and } \delta^t = \frac{\delta}{\sqrt{t}}, \tag{11}$$

where σ and δ are the initial bounded value and threshold, t is the clock of a *worker* node, and $\sqrt{t}$ works as a regularizer. In particular, it increases δ^t during the early stage of the training, dropping more *updates* to reduce communication cost. At a later stage, the regularizer decreases δ^t, keeping more tiny *updates* for network transmission. In Section 4, we show that the time-varying δ^t is important to guarantee the convergence of data-parallel ML algorithms.

Network-Aware. DVF allows *worker* nodes and *server* nodes to adjust δ^t based on the real-time network performance during the push and broadcast operations. In shared clusters, a PS-based ML application usually shares cluster resources, including bandwidth, with other applications like Spark jobs. In this case, the available bandwidth between a *worker* node and a *server* node is varying. For example, when a short-term Spark job is shuffling its intermediate results, the performance of a PS-based ML application could be affected due to bandwidth sharing. To address this problem, DVF allows *worker* nodes and *server* nodes to adaptively adjust δ^t based on the real-time network performance. Specifically, DVF uses the Self-Loading Periodic Streams (SLoPS) method [26] and its implementation *Pathload* to periodically measure the available bandwidth between two nodes. If the measured value is higher than a given minimum bandwidth requirement (for example 50Mbps), the network connection is "sufficient". Otherwise, it is "insufficient". We predefine two initial thresholds, δ_{low} and δ_{up}, for each type of network performance, where $\delta_{low} < \delta_{up}$. During the push operations, if the network connections between i-th node and all nodes it sends data to are "sufficient", i-th node would use δ_{low} as its initial threshold and $\delta_i^t = \delta_{low}/\sqrt{t}$. Otherwise, $\delta_i^t = \delta_{up}/\sqrt{t}$. In this way, *worker* nodes drop more *updates* when the network performance is not good, and vice versa. During the broadcast, *server* nodes would use δ_{low} if all network connections in the overlay network, which is detailed in Section 3.4, are "sufficient".

3.3.4 DVF Analysis

Worker nodes and *server* nodes need additional computation time to filter, compress and decompress *updates* during the push and broadcast operations. This additional computation overhead would not affect application performance much. Libraries like BLAS/Lapack provide sufficient computation capacity to perform simple filtering operations with multi-thread and SIMD. According to our experiments, snappy can compress and decompress a sparse vector at a rate of up to 950 MB/s per CPU core (2.4 GHZ), which would not be the performance bottleneck.

Each *worker* node needs additional memories to store all dropped *updates*. Since DVF aggregates dropped *updates* locally, each *worker* node uses the same amount of memories to store dropped *updates* as *parameters* at most. This addition memory usage would also not be the performance bottleneck. For example, AlexNet contains 61M *parameters*. If we use Float32 to represent each *parameter* and *update*, a *worker* node needs up to 233MB memory to store dropped *updates*. Current commodity server has sufficient memories to store drooped *updates*.

3.4 Tree-based Streaming Broadcast

We design TSB to optimize the performance of broadcast operations. With the conventional method, the limited network bandwidth of a *server* node would be shared by all n *worker* nodes, making it to be a potential performance bottleneck. To address this problem, TSB adopts a tree-based overlay network to propagate aggregated *updates* from the root nodes to the leaf nodes, thus improving the broadcast operation performance, as shown in Figure 6. While TSB may generate some additional communication overhead: a *worker* node needs more hops to get aggregated *updates*, we can limit such additional overhead by controlling the depth of the constructed tree. Compared to existing MPI-based broadcast libraries shown in [27], [16], [28], TSB is built over ZMQ to achieve data streaming from multiple roots.

3.4.1 Constructing a Tree-based Overlay Network

It is a challenging issue to determine the "best" topology for the tree-based overlay network in TSB to have minimum overall communication cost. As shown in Figure 6, a PS-based ML application could be deployed in a cluster with a common three-tier tree topology. Its *server* nodes and *worker* nodes can be located on any physical servers of the cluster. Previous studies have shown that communication cost between two nodes differs a lot for different locations in the cluster [29]. Take the tier-tree topology as an example, if two nodes are in same edge layer, the communication cost between them is low due to sufficient bandwidth. However, if they are in different aggregation layer, the communication cost becomes high due to the limited bandwidth between them. Let $c_{i,j}$ denote the communication cost (which is measured by network bandwidth, higher bandwidth means lower cost) between i-th node and j-th node of a PS-based ML application in a cluster. In this paper, based on previous measurements in [29], $c_{i,j}$ could be set to a predefined value as follows:

- $c_{i,j} = 0.1$, if node i, j are in the same edge layer;
- $c_{i,j} = 0.5$, if node i, j are in different edge layers and the same aggregation layer;
- $c_{i,j} = 1.0$, if node i, j are in different aggregation layers.

To construct a good tree-based overlay network for the broadcast operation, we propose an optimization-oriented solution with the objective of obtaining minimum communication cost. Consider a PS-based system with $n+p$ nodes, where the first n nodes are *worker* nodes and the rest p nodes are *server* nodes. $l_{i,j} = 1$ denotes that i-th node would directly send data to j-th node, and $l_{i,j} = 0$ otherwise. $t_i = 1$ denotes that i-th node is a leaf node in the tree-based overlay, and $t_i = 0$ otherwise. $h^k_{i,j} = 1$ denotes that the path from the root node to k-th node contains the connection from i-th node to j-th node. We can formulate the tree-based overlay network construction problem as a binary integer programming (BIP) problem as follows:

$$\textbf{P3:} \quad \min \sum_{i=1}^{n}\sum_{j=1}^{n} l_{i,j}c_{i,j} + \sum_{i=n+1}^{n+p}\sum_{j=1}^{n} l_{n+1,j}c_{i,j} + \alpha\sum_{i=1}^{n+1} t_i, \tag{12}$$

$$\text{s.t.} \sum_{i=1}^{n+1} l_{i,j} = 1, \qquad \forall j \in \{1,2,\ldots,n\} \tag{13}$$

$$\sum_{j=1}^{n+1} l_{i,j} \le T, \qquad \forall i \in \{1,2,\ldots,n+1\} \tag{14}$$

$$\sum_{j=1}^{n+1} l_{i,j} + t_i > 0, \qquad \forall i \in \{1,2,\ldots,n+1\} \tag{15}$$

$$\sum_{i=1}^{n+1}\sum_{j=1}^{n+1} h^k_{i,j} \le H, \ \forall k \in \{1,2,\ldots,n+1\} \tag{16}$$

$$h^k_{i,j} - h^{k'}_{i,j} \le 2(1 - l_{k,k'}), \quad i \ne k, j \ne k', \quad \forall i,j,k,k' \in \{1,2,\ldots,n+1\} \tag{17}$$

$$l_{i,n+1} = 0, \qquad \forall i \in \{1,2,\ldots,n+1\} \tag{18}$$

$$l_{i,i} = 0, \qquad \forall i \in \{1,2,\ldots,n+1\} \tag{19}$$

$$h^j_{i,j} = l_{i,j}, \qquad \forall i,j \in \{1,2,\ldots,n+1\} \tag{20}$$

$$t_i, l_{i,j}, h^k_{i,j} \in \{0,1\}, \quad \forall i,j,k \in \{1,2,\ldots,n+1\} \tag{21}$$

where T and H are two predefined integers denoting the maximum degree and the maximum depth of the tree-based overlay, respectively; $l_{i,j}$, t_i and $h^k_{i,j}$ are the binary decision variables. In the constraints, we only consider n *worker* nodes and 1 *server* node, since all *server* nodes would have same behavior: $l_{i,j} = l_{n+1,j}, \forall i \in \{n+1,\ldots,n+p\}, \forall j \in \{1,2,\ldots,n\}$. **P3** is orgnized as follows:

1) **Parent Node Constraints.** Equation 13 and 18 denote that each *worker* node only has one parent node, and all *server* nodes must be the root nodes. Equation 19 implies that a node cannot send data to itself.
2) **Child Node Constraints.** Equation 14 denotes that all nodes can have at most T child nodes. Equation 15 shows whether i-th node is a leaf node: if i-th node contains has no child nodes, $t_i = 1$, and vice versa (since t_i is a term in the objective function).
3) **Maximum Depth Constraints.** Equation 16 means that the maximum depth of the tree is no greater than a predefined integer H. In particular, $\sum_{i=1}^{n+1}\sum_{j=1}^{n+1} h^k_{i,j}$ denotes the depth of k-th node. Equation 17 and 20 implies that if $l_{k,k'} = 1$, k'-th node should contain all the path information of k-th node and $h^{k'}_{k,k'} = 1$.
4) **Objective Function.** The goal of Equation 12 is to minimize the overall communication cost of the constructed tree-based overlay network. Moreover, it penalizes the number of leaf nodes to balance the tree. For example, we can balance a tree with 6 worker nodes via minimizing the number of leaf nodes to 3 in Figure 7.

As we can see, **P3** is a generic BIP problem, which can be solved by BIP solvers like CPLEX.

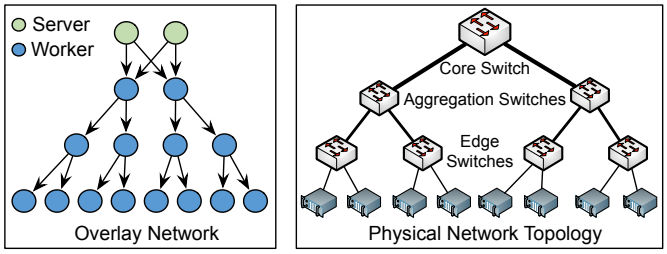


Fig. 6. Building a tree-based broadcast system for distributed ML in a cluster with tier-tree topology.

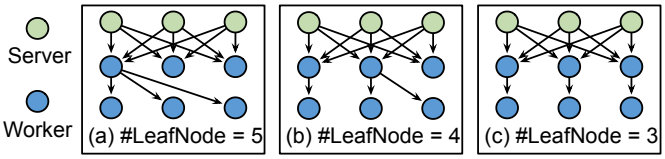


Fig. 7. Balancing a tree via penalizing the number leaf nodes.

3.4.2 Streaming Aggregated Updates

With the tree-based overlay network, TSB can efficiently broadcast aggregated *updates*, which are processed by DVF, from *server* nodes to *worker* nodes. For each broadcast vector $\boldsymbol{U}^t$ (which is filtered and compressed to a string by DVF), *server* nodes transmit it to *worker* nodes that it directly connects to, as shown in Figure 6. Whenever a *worker* node receives a block of data from its parent node, the node would transmit received data to its child nodes if it is a non-leaf node, and decompresses it. In this way, all *worker* nodes receive a copy of aggregated *updates*.

4 Convergence Analysis

In this section, we theoretically analyze PF, and use stochastic gradient descent (SGD) as an example to show how PF affect the convergence of distributed ML algorithms with both BSP and SSP. PF has UCC, DVF and TSB. Among them, UCC and TSB are purely system-level optimization. In contrast, DVF selectively drops *updates* for both push and broadcast operations; so workers might not be using the latest *updates* to compute a new set of *parameters*. This strategy may affect the convergence properties of some distributed ML algorithms. Hence, in this section, we focus on analyzing DVF's convergence guarantees. The analysis follows previous work such as [17], [21], [22], [30].

As in [21], [22], we simplify the formulation of an ML problem as follows: $J(\boldsymbol{W}) = \sum_{i=1}^{m} f_i(\boldsymbol{W})$, where $f_i(\boldsymbol{W}) = f(\boldsymbol{W}, D^i)$ is a predefined loss function on i-th example of the training dataset. With SGD, we use $\mathcal{D}^t$ to compute $\boldsymbol{W}^t$ at t-th iteration. In this case, we suffer the loss $f_t(\boldsymbol{W}^t)$. In the future iterations, we improve the predicted *parameters*. We measure the quality of the predicted *parameters* using *regret*, which is defined as follows:

$$R[\boldsymbol{W}] := \sum_{i=1}^{T}(f_t(\boldsymbol{W}^t) - f_t(\boldsymbol{W}^*)), \tag{22}$$

where $\boldsymbol{W}^*$ is the optimum solution. In this section, we restrict our discussion to convex and L-Lipschitz loss function $f(\cdot)$. As shown in [17], [21], [22], if $R[\boldsymbol{W}]/T \rightarrow 0$, our predicted $\boldsymbol{W}^t$ would converge to the optimum $\boldsymbol{W}^*$.

4.1 PF-BSP

In PF-BSP, we use following notations: $\boldsymbol{U}_i^t$ denotes the computed update vector on i-th *worker* node; $\boldsymbol{U}_{i,drop}^t$ is the dropped update vector on i-th *worker* node; $\boldsymbol{U}_{i,rmn}^t$ is pushed to *server* nodes; $\boldsymbol{U}^t = \sum_{i=1}^{n} \boldsymbol{U}_{i,rmn}^t$ denotes the aggregated *updates* on the *server* nodes; $\boldsymbol{U}_{drop}^t$ is the dropped update vector on the *server* nodes; $\boldsymbol{U}_{rmn}^t$ is broadcasted to *worker* nodes. *Worker* nodes only have "noisy" *parameters* due to the dropped *updates*, and $\hat{\boldsymbol{W}}^{t+1} = \hat{\boldsymbol{W}}^t - \boldsymbol{U}_{rmn}^t$. Let $\boldsymbol{W}^t$ denote the vector of "true" *parameters* produced by generic BSP, which can see effects of all computed *updates*. Assuming $t \geq 1$, so that: $\hat{\boldsymbol{W}}^t = \boldsymbol{W}^0 - \sum_{t'=0}^{t-1} \boldsymbol{U}_{rmn}^{t'}$, and $\boldsymbol{W}^t = \boldsymbol{W}^0 - \sum_{t'=0}^{t-1}\sum_{i=0}^{n} \boldsymbol{U}_i^{t'}$.

Lemma 1. Let $\boldsymbol{A}^t = \hat{\boldsymbol{W}}^t - \boldsymbol{W}^t$ denote the difference between $\hat{\boldsymbol{W}}^t$ and $\boldsymbol{W}^t$. It follows that:

$$||\boldsymbol{A}^t||_2 \leq (n+1)\delta^t\sqrt{d}, \tag{23}$$

where d is the size of $\boldsymbol{W}$; n is the number of *worker* nodes; and δ^t is the threshold of DVF at t-th iteration.

Lemma 1 implies that PF-BSP guarantees a bound on the difference between the "noisy" *parameters* and the "true" *parameters*. It is an important property to show the convergence of PF-BSP. We show the proof in Appendix A.

Theorem 1. Given a function $\sum_{i=1}^{T} f_i(\boldsymbol{W})$, we use SGD to search for $\boldsymbol{W}^*$ with learning rate $\eta^t = \eta/\sqrt{t}$. DVF with threshold $\delta^t = \delta/\sqrt{t}$ selectively drops some entries of the computed update vector $\boldsymbol{U}^t$, where $\boldsymbol{U}^t = \eta^t \nabla f_t(\hat{\boldsymbol{W}}^t)$. Under suitable conditions ($f_t(\cdot)$ is convex and L-Lipschitz, and the distance between any two points $D(\boldsymbol{W}_1||\boldsymbol{W}_2) = \frac{1}{2}||\boldsymbol{W}_1 - \boldsymbol{W}_2||_2^2 \leq F^2$), we get the *regret* as follows:

$$\begin{aligned} R[\boldsymbol{W}] &:= \sum_{t=1}^{T} f_t(\hat{\boldsymbol{W}}^t) - f_t(\boldsymbol{W}^*) \\ &\leq \eta L^2\sqrt{T} + \frac{F^2\sqrt{T}}{\eta} + 2\delta(n+1)L\sqrt{dT} \\ &= \mathcal{O}(\sqrt{T}), \end{aligned} \tag{24}$$

where F and L are constants.

Theorem 1 implies that $\hat{\boldsymbol{W}}$ converges to the optimum $\boldsymbol{W}^*$ with PF-BSP, since $R[\boldsymbol{W}]/T \rightarrow 0$. If the initial threshold of DVF δ is equal to 0, PF-BSP reduces to native BSP with the same *regret* as in [17]. To guarantee convergence, the value of δ^t should decrease over time. We show the proof of Theorem 1 in Appendix A.

4.2 PF-SSP

With PF-SSP, a *worker* node has to use noisy *parameters* $\hat{\boldsymbol{W}}^t$ to perform computations for two reasons. First, as with PF-BSP, $\hat{\boldsymbol{W}}^t$ cannot see the effects of dropped *updates*. Second, as with generic SSP, $\hat{\boldsymbol{W}}^t$ can see the effects of additional *updates* with timestamp greater than t pushed by some fast *worker* nodes. Compared to generic SSP, PF-SSP makes all *worker* nodes have same view of *parameters*, since PF-SSP uses the broadcast operation to make *worker* nodes update *parameters* locally. Assuming $t \geq 1$, so that $\hat{\boldsymbol{W}}^t = \boldsymbol{W}^0 - \sum_{t'=0}^{t-1} \boldsymbol{U}_{rmn}^{t'}$, and $\boldsymbol{W}^t = \boldsymbol{W}^0 - \sum_{t'=0}^{t-1}\sum_{i=0}^{n} \boldsymbol{U}_i^{t'}$, where $\boldsymbol{U}_{rmn}^{t'}$ denotes the broadcasted aggregated *updates*; and $\boldsymbol{W}^t$ denotes the vector of "true" *parameters* produced by generic BSP.

Lemma 2. Let $\boldsymbol{A}^t = \hat{\boldsymbol{W}}^t - \boldsymbol{W}^t$ denote the difference between $\hat{\boldsymbol{W}}^t$ and $\boldsymbol{W}^t$ with PF-SSP. Under the same conditions as in Theorem 1, it follows that:

$$||\boldsymbol{A}^t||_2 < (n+1)\delta^t\sqrt{d} + ns\eta^t L, \tag{25}$$

where d is the size of $\boldsymbol{W}$; L is a constant.

Lemma 2 implies that PF-SSP guarantees a bound on the difference between the "noisy" *parameters* and the "true" *parameters*. We show the proof of Lemma 2 in Appendix A.

Theorem 2. Given a function $\sum_{i=1}^{T} f_i(\boldsymbol{W})$, we use SGD to search for $\boldsymbol{W}^*$ with learning rate $\eta^t = \eta/\sqrt{t}$. DVF with threshold $\delta^t = \delta/\sqrt{t}$ selectively drops some entries of the computed update vector $\boldsymbol{U}^t$, where $\boldsymbol{U}^t = \eta^t \nabla f_t(\hat{\boldsymbol{W}}^t)$. The staleness threshold of PF-SSP is s. Under same conditions as in Theorem 1, we get the *regret* as follows:

$$\begin{aligned} R[\boldsymbol{W}] &:= \sum_{t=1}^{T} f_t(\hat{\boldsymbol{W}}^t) - f_t(\boldsymbol{W}^*) \\ &\leq \eta L^2\sqrt{T} + \frac{F^2\sqrt{T}}{\eta} + 2\delta(n+1)L\sqrt{dT} + 2\eta ns L^2\sqrt{T} \\ &= \mathcal{O}(\sqrt{T}). \end{aligned} \tag{26}$$

where F and L are constants.

Theorem 2 implies that $\hat{\boldsymbol{W}}$ converges to the optimum $\boldsymbol{W}^*$ with PF-SSP, since $R[\boldsymbol{W}]/T \rightarrow 0$. If the initial threshold of DVF δ is equal to 0, PF-SSP reduces to a variant of SSP. To guarantee converge, PF-BSP also needs to decrease δ^t over time. We show the proof of Theorem 2 in Appendix A.

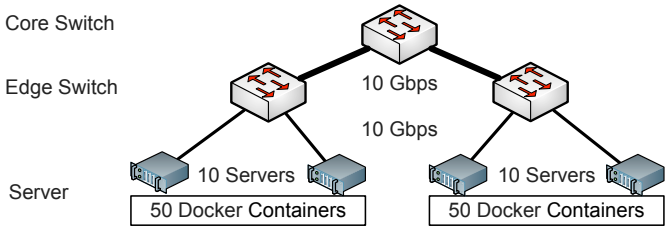


Fig. 8. We use a testbed with a tier tree topology to evaluate PF.

5 Numerical Results and Analysis

In this section, we evaluate PF's performance using a working testbed. We first investigate PF's impact on network traffic. Then, we analyze the communication time and computation time of a PF-enabled distributed ML system. Next, we show PF's performance improvement.

5.1 Experiments' Configuration

5.1.1 Hardware

We use a testbed with a tier tree topology to conduct experiments. As shown in Figure 8, the testbed has 20 physical servers running up to 100 Docker containers. Each physical server contains two Intel Xeon E5-2600 CPUs, 128GB memory and 4TB HDD. The testbed contains 5 NVIDIA Tesla K40 GPU accelerators, which are assigned to 5 containers. All physical network connections are 10Gbps links. Each container is allocated with 4 CPU cores, 20GB memory, and 1 Gbps network bandwidth.

5.1.2 Software

We use Python and C++ to implement a lightweight PF-enabled distributed ML system, and make it work in conjunction with several ML engines, including MxNet, Theano and Caffe. Specifically, MxNet[4], Theano and Caffe compute *updates* on *worker* nodes. PF exchanges data between *worker* nodes and *server* nodes. We also implement two synchronization models on the PF-enabled distributed ML system: BSP and SSP-n, where n is the staleness threshold.

We use PS-lite, which is an implementation of the generic PS framework and a core module in MxNet, as our baseline system. Compared to our PF-enabled distributed ML system, the baseline system has the same computation performance, since it also uses MxNet, Theano [31] and Caffe [32] on *worker* nodes to perform computation. The key difference is that the baseline system uses conventional push-pull communication model to exchange data between *worker* nodes and *server* nodes. It should be noted that the main target of PF is to improve the network performance of distributed ML on the PS framework in commodity clusters. While MPI and its implementations (e.g., MVAPICH2, OpenMPI) also improve the network performance of distributed ML, we do not compare PF with them in the experiments for three reasons: 1) MPI and PF use different system architectures for distributed ML. Specifically, MPI uses the P2P architecture, while PF improves the network performance for the PS framework. 2) MPI requires fast network fabrics to maximize its performance in HPC clusters. However, only commodity network is available in our testbed. 3) Existing MPI-based approaches usually optimize deep learning applications on GPU with CUDA-Aware techniques, which require fast network fabrics. PF is designed to reduce the communication overhead for generic distributed ML applications on both CPU and GPU.

4. MxNet supports distributed ML applications. In our experiments, we use MxNet in single-node mode to compute *updates* on *worker* nodes.

TABLE 2
Used ML models and datasets.

Model	Model Size	Training Dataset	Batch	Engine
LR	105M Float32	Criteo Subset 680M ad clicks	10M	CPU MxNet
MF	16.3M Float32	MovieLens 22M movies ratings	1M	CPU Theano
AlexNet	61M Float32	ImageNet Subset 100k labeled images	128	GPU Caffe

5.1.3 ML Models, Datasets and Configurations

In the experiments, we train 3 popular ML models, Logistic Regression (LR), Matrix Factorization (MF) and AlexNet, using publicly available datasets. In Table 2, we summarize model size, training dataset, batch size and the ML engine used for each model. To train LR and MF, we use 40 CPU *worker* nodes and 10 *server* nodes. To train AlexNet, we use 5 GPU *worker* nodes and 5 *server* nodes. Each *server* and *worker* node is placed on a container.

For each model, we use the following values of δ_{up} and δ_{low}, which are the thresholds in DVF: (i) LR, $\delta_{up} = 2\times10^{-4}$, $\delta_{low} = 1\times10^{-4}$; (ii) MF, $\delta_{up} = 1\times10^{-3}$, $\delta_{low} = 5\times10^{-4}$; and (iii) AlexNet, $\delta_{up} = 5\times10^{-3}$, $\delta_{low} = 2\times10^{-3}$. These values are derived from history log files. Typically, DVF with δ_{up} roughly drops 60% to 80% of *updates* on average per push or broadcast operation. The corresponding value for DVF with δ_{low} ranges from 40% to 60%.

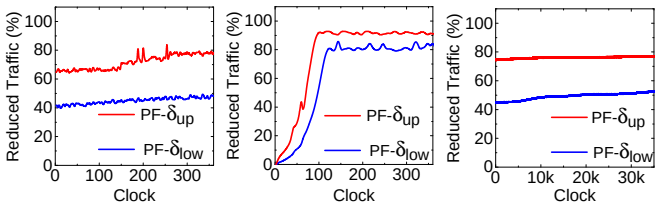
To further evaluate PF's performance, we run experiments in both a dedicated and a shared cluster. In a dedicated cluster, no other applications compete for bandwidth. In a shared cluster, each distributed ML application would share cluster resources with a set of Spark jobs. In particular, we set up a Spark cluster with 50 Docker containers, and periodically submit 50 Spark WordCount jobs. Spark jobs would generate huge volumes of network traffic during the data shuffling phase. We use Dorm [33] to manage different systems and applications in the shared cluster.

5.1.4 Performance Metrics

We collect the following data to measure PF's performance: (i) *Effect of DVF*, we measure the percentage of reduced network traffic when training ML models in a dedicated cluster; (ii) *Effect of TSB*, we measure the percentage of reduced time used to transmit aggregated *updates* in a dedicated cluster; (iii) *Communication Time vs. Computation Time*, we measure the communication time and computation time used to train ML models in a dedicated cluster; and (iv) *Speedup Ratio*, we measure ML model training speed in a dedicated cluster and a shared cluster.

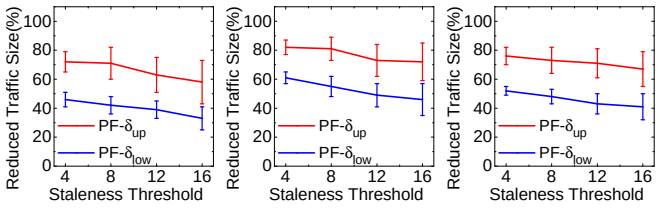
5.2 Effect of DVF

To evaluate DVF, we measure the percentage of reduced network traffic when training ML models with PF-BSP



(a) LR (b) MF (c) AlexNet

Fig. 9. Percentage of reduced network traffic with PF-BSP.



(a) LR (b) MF (c) AlexNet

Fig. 10. Percentage of reduced network traffic with PF-SSP.

and PF-SSP to achieve the same convergence criteria in a dedicated cluster in this set of experiments.

5.2.1 Reduced Network Traffic of PF-BSP

As shown in Figure 9, PF-BSP could reduce a large portion of network traffic for the three ML models used in our experiments, since DVF in PF selectively drop *updates* during the push and broadcast operations. In particular, compared to the baseline system, PF-BSP could reduce up to 80% and 50% of network traffic when training the LR model with δ_{up} and δ_{low}, respectively. For MF, PF-BSP could roughly reduce up to 90% and 82% of network traffic with δ_{up} and δ_{low} after clock 30, respectively. The corresponding values for AlexNet with δ_{up} and δ_{low} are 80% and 45%, respectively.

5.2.2 Reduced Network Traffic Under PF-SSP

Figure 10 shows that PF-SSP could also reduce a large portion of network traffic for the three ML models. Compared to the baseline system with SSP, PF-SSP could reduce up to 70%, 80% and 78% of network traffic when training LR, MF, and AlexNet with δ_{up}. The corresponding values for δ_{down} are 46%, 61% and 52%, respectively. From Figure 10 , we observe that the percentage of reduced network traffic decreases when using higher staleness thresholds. For example, when training AlexNet with δ_{low}, PF-SSP-16 roughly reduces 40% of network traffic. The corresponding value for PF-SSP-4 is 52%. According to our analysis, it is mainly caused by the fact that PF-SSP allows faster *worker* nodes to push more *updates* to *server* nodes. This would increase the number of *updates* retained for transmission. Still, PF-SSP could help to reduce a large portion of network traffic regardless of the staleness threshold.

5.3 Effect of TSB

To evaluate TSB, we measure the communication time used by a *worker* node to receive all *aggregated* updates from the *server* node per broadcast operation with TSB and DVF-TSB, and compare to the case without TSB. Without TSB, *worker* nodes pull data from *server* nodes individually. In this set of experiments, the maximum degree of the tree in TSB is set to 5, and the maximum depth is 3.

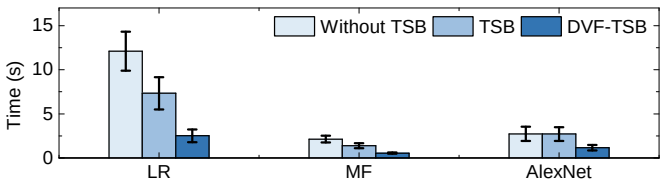


Fig. 11. Communication time used by a *worker* node to receive all *aggregated* updates per broadcast operation.

As shown in Figure 11, TSB could significantly reduce the communication time to propagate aggregated *updates*. In this set of experiments, DVF is disabled. Compared to the case without TSB, TSB could reduce the communication time by a factor of about 1.7 and 1.6 for LR and MF, respectively. Since our GPU testbed only has 5 *worker* nodes, TSB has a similar performance to the case without TSB when training AlexNet, since they have same architecture and communication behavioures.

When working in conjunction with DVF, TSB could further reduce the communication time to propagate aggregated *updates*. Compared to the case without TSB, DVF-TSB could reduce the communication time by a factor of about 4.8, 3.6 and 2.3 for LR, MF and AlexNet, respectively. The performance gain comes from the reduced network traffic and the optimized tree-based overlay network together.

5.4 Communication Time vs. Computation Time

In this set of experiments, we measure the communication time and computation time (which includes the compression and decompression time) used to train LR, MF and AlexNet with PF and the baseline system to achieve the same convergence criteria.

As shown in Figure 12, PF could significantly reduce the communication time used to train the three ML models when working in conjunction with both BSP and SSP. For LR, the baseline system with BSP takes about 18.3h on communication, which includes the time for pulling *parameters* and pushing *updates*. In contrast, PF-BSP uses around 4.8h and 8.1h on communication with δ_{up} and δ_{low}, respectively. Compared to the baseline system with BSP, PF-BSP improves the performance on communication by a factor of up to 3.8 to train LR. The corresponding improvement factors for MF and AlexNet are 4.1 and 5.4, respectively. PF-BSP with δ_{up} has better communication performance than δ_{low}, since DVF with δ_{up} could drop more *updates* and reduce more communication time correspondingly.

From Figure 12, we observe that PF-SSP also has significant performance improvement on communication. As shown in Figures 12(a-1)(b-1)(c-1), though the baseline system with SSP has better communication performance than the case with BSP, its communication overhead is still significant. For example, when training AlexNet with SSP-16, the baseline system roughly consumes 2.3 times more time on communication than computation. In contrast, Figures 12(a-2)(a-3)(b-2)(b-3)(c-2)(c-3) show that PF-SSP can reduce the communication time to a negligible value with large

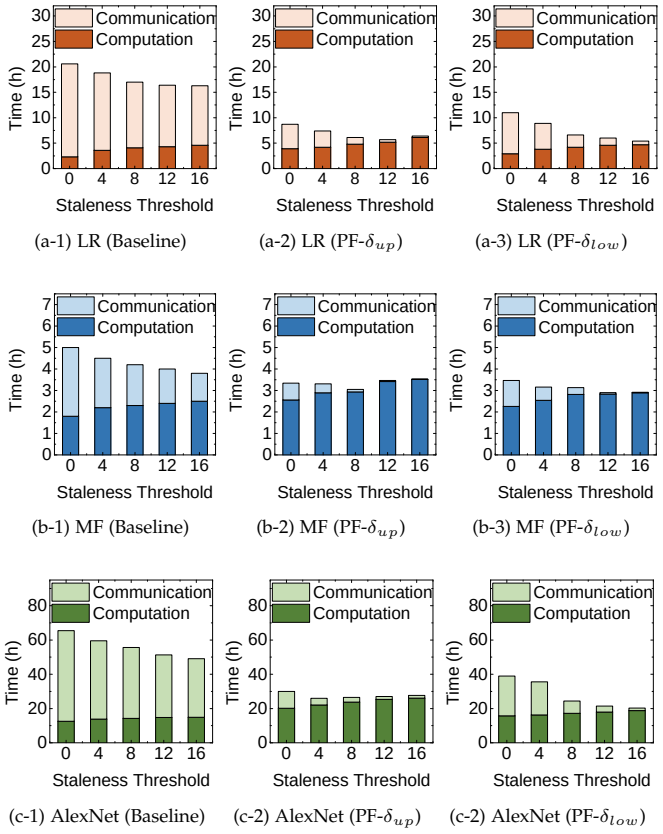


Fig. 12. Communication and computation time used to train LR, MF and AlexNet with various staleness threshold (threshold of 0 denotes BSP).

enough staleness threshold (for example 16). Such significant performance improvement is possible since PF and SSP can work together to efficiently overlap communication with computation. As a result, most of push and broadcast operations are performed during computation.

From Figure 12, we find that PF reduces the communication time at the cost of higher computation time. For example, when training LR, MF and AlexNet with δ_{up}, PF-BSP could increase computation time by a factor of up to 1.68, 1.42 and 1.75 respectively, compared to the baseline system with BSP. The corresponding values for δ_{low} are 1.22, 1.35 and 1.28. However, due to the significant performance gain on communication, the additional computation overhead would not affect the overall training time much. It should be noted that PF would not affect the achieved objective value of PS-based ML applications. From Figure 12, we can see that PF-BSP could reduce the overall training time by a factor of up to 2.7, 1.4 and 2.2 for LR, MF and AlexNet respectively, compared to the baseline system with BSP. The corresponding improvement factors with PF-SSP-16 are 2.5, 1.3 and 3.5, compared to the baseline system with SSP-16.

5.5 Speedup Ratio

In this set of experiments, we measure the speedup ratio of PF when training ML models in a dedicated and a shared cluster. We measure the time used by PF-BSP, PF-SSP and the baseline-system-SSP to achieve the convergence criteria for LR, MF and AlexNet. We define the speedup ratio as the total time obtained in PF-BSP, PF-SSP or baseline-system-SSP divided by the time obtained in baseline-system-BSP.

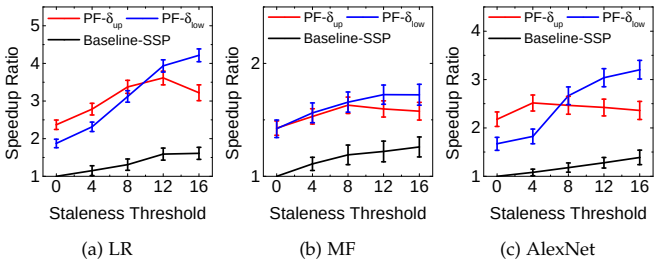


Fig. 13. Speedup ratio of PF in a dedicated cluster.

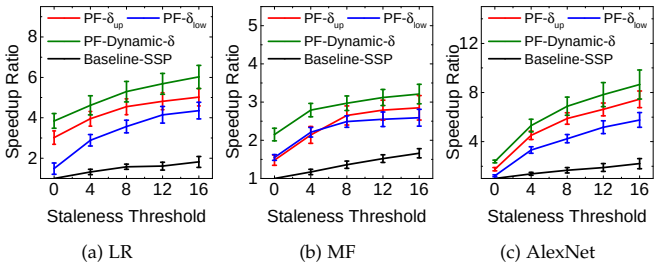


Fig. 14. Speedup ratio of PF in a shared cluster.

5.5.1 Dedicated Cluster

As shown in Figure 13, PF can speed up PS-based ML applications considerably. Specifically, PF-BSP can reduce the training time by a factor of up to 2.4, 1.4, and 2.2 for LR, MF and AlexNet, respectively. The baseline system with SSP could only speed up the training task by a factor of up to 1.6, 1.3 and 1.4. In contrast, the corresponding measurements for PF-SSP are 4.3, 1.7 and 3.2.

From Figure 13, we can observe that PF with δ_{low} may have a better or worse performance compared to the case with δ_{up}. It is due to the trade-off between reduced communication overhead and additional computation overhead. If the staleness threshold is low, the faster *worker* nodes are more likely to be blocked and wait for the slower *worker*. In this case, communication may stall computation. Thus, it is important to reduce communication time with δ_{up}, which could reduce the communication time to avoide the waiting operations. When the staleness threshold is high (for example 16), PF with δ_{low} can have a similar performance on communication to PF with δ_{up}. In this case, PF with δ_{low} has higher overall performance than PF with δ_{up} due to a smaller amount of additional computation overhead .

5.5.2 Shared Cluster

As shown in Figure 14, PF could also speed up PS-based ML applications significantly. When using the fixed threshold δ_{low}, PF could speed up the training task by a factor of up to 4.5, 2.6 and 3.8 for LR, MF and AlexNet, respectively. Since communication becomes the primary performance bottleneck in a shared cluster due to the limited network bandwidth, PF with δ_{up} has better performance than the case with δ_{low} most of the time. When dynamically adjusting δ based on real-time network performance, PF could achieve a higher speedup ratio compared to the case with fixed thresholds. Specifically, PF with dynamic thresholds could speed up PS-based ML applications by a factor of up to 6.1, 3.5 and 8.2 for LR, MF and AlexNet, respectively.

6 Related Work

In this section, we discuss a number of approaches to reduce communication overhead for distributed ML applications.

Reducing Communication Frequency. DistBelief [34], Petuum [4] and SparkNet [35] could reduce the communication frequency for distributed ML applications. In particular, DistBelief and Petuum are based on the PS framework; and SparkNet uses the iterative MapReduce framework. DistBelief allows *worker* nodes to pull latest *parameters* every u steps and push generated *parameters* every v steps, where u might not be equal to v. In Petuum, *worker* nodes could use cached stale *parameters* to compute *updates*, until the version of cached *parameters* is older than a threshold. In SparkNet, every *worker* node performs τ iterations of computations at each step, after which all *parameters* are aggregated, averaged and broadcasted to *worker* nodes. PF is a complementary approach; it does not alter the communication frequency. Users can define and optimize the frequency of their push/broadcast operations easily when using with PF.

High-Performance Network Fabrics. S-Caffe [11], FireCaffe [36], Malt [12] and COTS-HPC [13] use InfiniBand and MPI to improve the performance of distributed ML in HPC clusters. To further improve GPU-GPU communication, several MPI implementations, such as MVAPICH2 and OpenMPI provide efficient CUDA-Aware support using techniques like GPUDirect RDMA [11], [15], [16]. Compared to these MPI-based approaches, which uses a P2P architecture, PF is designed to improve the network performance of generic PS-based distributed ML applications in commodity clusters with restricted network bandwidth.

Partial Communication. Li-PS [18] and Ako [37] can selectively send out only parts of *updates*. In particular, Li-PS is an implementation of the generic PF framework; and Ako is a decentralized system without *server* nodes. Li-PS leverages user-defined filters like Karush-Kuhn-Tucker (KKT) filter to select *updates*, which are likely to have an effect on the corresponding *parameters*, for transmission. In Ako, a *worker* node sends out only one partition of computed *updates* to other peer nodes in a single iteration, with the remaining partitions transmitted in subsequent iterations. PF adopts a similar but different approach to Li-PS: Li-PS removes unselected *updates* for transmission, thus it only works with some particular applications with sparse features; PF would not lose any update information by aggregating dropped *updates* in the following iteration, and it can reduce the communication overhead for generic ML applications.

Communication/Computation Overlapping. Petuum, Bösen [38] and Poseidon [20] could overlap communication with computation. Petuum and Bösen leverage the SSP synchronization model to overlap the communication of previous iterations with the computation of current iteration. In addition, Bösen also prioritizes communication of fast-changing *parameters*. Poseidon is a deep learning platform, which overlaps the communication of top layers with the computation of bottom layers of a deep neural network in a single iteration. We have shown that PF could work in conjunction with SSP and achieve a higher performance gain due to communication/computation overlapping.

Model Compression. Poseidon [20], CNTK [39], Bösen and Gupta et al. [40] use several compression techniques to reduce the size of *updates* for network transmission. Poseidon uses sufficient factors [10] to represent dense fully connected layers of CNNs. CNTK represents *updates* as 1-bit values for speech deep neural networks (DNNs) with some negative impacts on model accuracy. Bösen and Gupta et al. study the effect of limited precision data representation on DNNs. They show that DNNs can be trained with 16-bit fixed-point numbers instead of 32-bit floating-point numbers, which would not reduce model accuracy. In this paper, PF uses 32-bit floating-point number representation to support common ML models. For some particular ML models, PF can also use 16-bit floating-point number representation to further reduce the communication overhead.

Partial Broadcasting. SFB [10], MALT [12] and Mariana [41] uses partial broadcasting to replace all-to-all broadcasting in decentralized distributed ML systems. Native broadcasting in a decentralized, peer-to-peer system would incur quadratic communication cost with respect to the number of nodes. To address this problem, SFB proposes a partial broadcasting system, in which a node only broadcast *updates* among a part of directly connected nodes. In MALT, nodes exchanges *updates* with a subset of nodes selected by a Halton sequence. Mariana manages multiple nodes workers in a linear topology, and only sends data to adjacent workers. In this paper, PF builds a tree-based overlay network to broadcast aggregated *updates* from *server* nodes to *worker* nodes. Compared to aforementioned approaches, PF is built on top of the more popular PS framework rather than a decentralized system.

Overlay Network. MLNet [19] uses an overlay network to aggregate *updates* on *worker* nodes, and send aggregated *updates* to *server* nodes during the push operations. PF also builds an overlay network to improve the performance of its broadcast operations. PF lets each *worker* node send *updates* to *server* nodes without aggregation for two reasons: 1) PF compresses all network-transmitted messages, thus *update* aggregation incurs a lot of compression and decompression delay. In contrast, TSB lets each *worker* node relay received messages to its child nodes without compressing them again. 2) It is difficult to implement BAP models with *update* aggregation, since *server* nodes cannot know the process of each *worker* node.

7 Summary

In this paper, we propose an approach named Parameter Flow (PF) to optimize network performance for distributed ML applications on the PS framework. When training big ML models with the conventional PS framework, *worker* nodes frequently pull *parameters* and push *updates*, resulting in high communication overhead. Our investigations showed that modern distributed ML applications could spend much more time on communication than computation. PF tackles this problem with three techniques: an update-centric communication (UCC) model, a dynamic value-bounded filter (DVF), and a tree-based streaming broadcasting (TSB) system. In particular, UCC introduces a broadcast/push model to exchange data between *worker* nodes and *server* nodes. DVF reduces network traffic and communication time by selectively dropping *updates* for network transmission. TSB optimizes the performance of the

broadcast operation via a tree-based overlay network. Extensive performance evaluations showed that PF could significantly reduce communication overhead for distributed ML applications on the PS framework. In the future, we plan to apply PF to more distributed ML applications.

APPENDIX A

A.1 Proof of Lemma 1

Compared to $\hat{\boldsymbol{W}}^t$, $\boldsymbol{W}^t$ could incorporate all dropped *updates*, thus $\boldsymbol{A}^t = \hat{\boldsymbol{W}}^t - \boldsymbol{W}^t = \sum_{i=1}^{n} \boldsymbol{U}_{i,drop}^{t-1} + \boldsymbol{U}_{drop}^{t-1}$. Then, the magnitude of $\boldsymbol{A}^t$ has an upper-bound,

$$||\boldsymbol{A}^t||_2 \leq \sum_{i=1}^{n} ||\boldsymbol{U}_{i,drop}^{t-1}||_2 + ||\boldsymbol{U}_{drop}^{t-1}||_2,$$

since $||\boldsymbol{a}+\boldsymbol{b}||_2 \leq ||\boldsymbol{a}||_2 + ||\boldsymbol{b}||_2$ for all $\boldsymbol{a}$ and $\boldsymbol{b}$. Since DVF guarantees a bound on the magnitude of dropped *updates*, we get $||\boldsymbol{U}_{i,drop}^{t-1}||_2 \leq \sigma^t, i = \{1, 2, \ldots, n\}$, and $||\boldsymbol{U}_{drop}^{t-1}||_2 \leq \sigma^t$, where σ^t is the bound value. Hence,

$$||\boldsymbol{A}^t||_2 \leq (n+1)\sigma^t = (n+1)\delta^t\sqrt{d}.$$

This completes the proof of Lemma 1. □

A.2 Proof of Theorem 1

Since $f_t(\cdot)$ is convex, $R[\boldsymbol{W}] := \sum_{t=1}^{T} f_t(\hat{\boldsymbol{W}}^t) - f_t(\boldsymbol{W}^*) \leq \sum_{t=1}^{T} \left\langle \nabla f_t(\hat{\boldsymbol{W}}^t), \hat{\boldsymbol{W}}^t - \hat{\boldsymbol{W}}^* \right\rangle$. Based on Lemma 2 in [22], we can expand $R[\boldsymbol{W}]$ as follows:

$$R[\boldsymbol{W}] \leq \eta L^2\sqrt{T} + \frac{F^2\sqrt{T}}{\eta} + \sum_{t=1}^{T} \left\langle \boldsymbol{A}^t, \nabla f_t(\hat{\boldsymbol{W}}^t) \right\rangle.$$

According to L-Lipschitz assumption, i.e. $||\nabla f_t(\hat{\boldsymbol{W}}^t)||_2 \leq L$, and Lemma 1, we get the upper-bound of the third term: $\sum_{t=1}^{T} \left\langle \boldsymbol{A}^t, \nabla f_t(\hat{\boldsymbol{W}}^t) \right\rangle \leq \sum_{t=1}^{T} (n+1)\delta^t L\sqrt{d} \leq 2\delta(n+1)L\sqrt{dT}$. Hence, we get

$$R[\boldsymbol{W}] \leq \eta L^2\sqrt{T} + \frac{F^2\sqrt{T}}{\eta} + 2\delta(n+1)L\sqrt{dT} = \mathcal{O}(\sqrt{T}).$$

This completes the proof of Theorem 1. □

A.3 Proof of Lemma 2

Compared to $\boldsymbol{W}^t$, $\hat{\boldsymbol{W}}^t$ contains additional *updates* from fast *worker* nodes, and misses dropped *updates*. Since additional *updates* are also processed by DVF in PF-SSP, so that $\boldsymbol{A}^t = \hat{\boldsymbol{W}}^t - \boldsymbol{W}^t = \sum_{i=1}^{n} \boldsymbol{U}_{i,drop}^{t-1} + \boldsymbol{U}_{drop}^{t-1} - \sum_{i\in\mathcal{B}} \sum_{\hat{t}=t}^{t+s-1} \boldsymbol{U}_i^{\hat{t}}$, where $\mathcal{B}$ is the set of fast *worker* nodes which push additional *updates*. According to Lemma 1, we get

$$\begin{aligned} ||\boldsymbol{A}^t||_2 &\leq (n+1)\delta^t\sqrt{d} + ||\sum_{i\in\mathcal{B}} \sum_{\hat{t}=t}^{t+s-1} \boldsymbol{U}_i^{\hat{t}}||_2 \\ &\leq (n+1)\delta^t\sqrt{d} + \sum_{i\in\mathcal{B}} \sum_{\hat{t}=t}^{t+s-1} ||\eta^{\hat{t}} \nabla f_{\hat{t}}(\boldsymbol{W}^{\hat{t}})||_2. \end{aligned}$$

Due to L-Lipschitz assumption, $||\nabla f_t(\hat{\boldsymbol{W}}^t)||_2 \leq L$. Since $\eta^t = \eta/\sqrt{t}$ and $|\mathcal{B}| < n$, we get

$$||\boldsymbol{A}^t||_2 < (n+1)\delta^t\sqrt{d} + ns\eta^t L.$$

This completes the proof of Lemma 2. □

A.4 Proof of Theorem 2

The proof of Theorem 2 is similar with the proof of Theorem 1. We expand $R[\boldsymbol{W}]$ as follows:

$$R[\boldsymbol{W}] \leq \eta L^2\sqrt{T} + \frac{F^2\sqrt{T}}{\eta^0} + \sum_{t=1}^{T} \left\langle \boldsymbol{A}^t, \nabla f_t(\hat{\boldsymbol{W}}^t) \right\rangle.$$

According to L-Lipschitz assumption ($||\nabla f_t(\hat{\boldsymbol{W}}^t)||_2 \leq L$) and Lemma 2, we get the upper-bound of the third term: $\sum_{t=1}^{T} \left\langle \boldsymbol{A}^t, \nabla f_t(\hat{\boldsymbol{W}}^t) \right\rangle \leq L \sum_{t=1}^{T} \left[(n+1)\delta^t\sqrt{d} + ns\eta^t L\right] \leq 2\delta(n+1)L\sqrt{dT} + 2\eta ns L^2\sqrt{T}$. Hence,

$$\begin{aligned} R[\boldsymbol{W}] &\leq \eta L^2\sqrt{T} + \frac{F^2\sqrt{T}}{\eta} + 2\delta(n+1)L\sqrt{dT} + 2\eta ns L^2\sqrt{T} \\ &= \mathcal{O}(\sqrt{T}). \end{aligned}$$

This completes the proof of Theorem 2. □

REFERENCES

[1] T.-S. Chua, X. He, W. Liu, M. Piccardi, Y. Wen, and D. Tao, "Big data meets multimedia analytics," *Signal Processing*, no. 124, pp. 1–4, 2016.

[2] H. Hu, Y. Wen, T.-S. Chua, and X. Li, "Toward scalable systems for big data analytics: A technology tutorial," *IEEE Access*, vol. 2, pp. 652–687, 2014.

[3] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *OSDI 14*, 2014, pp. 583–598.

[4] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu, "Petuum: a new platform for distributed machine learning on big data," *Big Data, IEEE Transactions on*, vol. 1, no. 2, pp. 49–67, 2015.

[5] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[6] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project adam: Building an efficient and scalable deep learning training system," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 571–582.

[7] W. Wang, G. Chen, A. T. T. Dinh, J. Gao, B. C. Ooi, K.-L. Tan, and S. Wang, "Singa: Putting deep learning in the hands of multimedia users," in *Proceedings of the 23rd ACM international conference on Multimedia*. ACM, 2015, pp. 25–34.

[8] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. GA: USENIX Association, Nov. 2016.

[9] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition (2014). arxiv preprint," *arXiv preprint arXiv:1409.1556*.

[10] P. Xie, J. K. Kim, Y. Zhou, Q. Ho, A. Kumar, Y. Yu, and E. Xing, "Lighter-communication distributed machine learning via sufficient factor broadcasting," in *Proceedings of the 32nd International Conference on Conference on Uncertainty in Artificial Intelligence*, 2016.

[11] A. A. Awan, K. Hamidouche, J. M. Hashmi, and D. K. Panda, "S-Caffe: Co-designing mpi runtimes and caffe for scalable deep learning on modern gpu clusters," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2017, pp. 193–205.

[12] H. Li, A. Kadav, E. Kruus, and C. Ungureanu, "Malt: distributed data-parallelism for existing ml applications," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 3.

[13] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with COTS HPC systems," in *Proceedings of the 30th international conference on machine learning*, 2013, pp. 1337–1345.

[14] I. T. Association *et al.*, *InfiniBand Architecture Specification: Release 1.0*. InfiniBand Trade Association, 2000.

[15] H. Wang, S. Potluri, M. Luo, A. K. Singh, S. Sur, and D. K. Panda, "MVAPICH2-GPU: optimized gpu to gpu communication for infiniband clusters," *Computer Science-Research and Development*, vol. 26, no. 3-4, p. 257, 2011.

[16] A. A. Awan, K. Hamidouche, A. Venkatesh, and D. Panda, "Efficient large message broadcast using NCCL and CUDA-Aware mpi for deep learning," in *Proceedings of the 23rd European MPI Users' Group Meeting*. ACM, 2016, pp. 15–22.

[17] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More effective distributed ML via a stale synchronous parallel parameter server," in *Advances in neural information processing systems*, 2013, pp. 1223–1231.

[18] M. Li, D. G. Andersen, A. J. Smola, and K. Yu, "Communication efficient distributed machine learning with the parameter server," in *Advances in Neural Information Processing Systems*, 2014, pp. 19–27.

[19] L. Mai, C. Hong, and P. Costa, "Optimizing network performance in distributed machine learning," in *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, 2015.

[20] H. Zhang, Z. Hu, J. Wei, P. Xie, G. Kim, Q. Ho, and E. Xing, "Poseidon: A system architecture for efficient GPU-based deep learning on multiple machines," *arXiv preprint arXiv:1512.06216*, 2015.

[21] W. Dai, A. Kumar, J. Wei, Q. Ho, G. Gibson, and E. P. Xing, "High-performance distributed ml at scale through parameter server consistency models," *arXiv preprint arXiv:1410.8043*, 2014.

[22] J. Wei, W. Dai, A. Kumar, X. Zheng, Q. Ho, and E. P. Xing, "Consistent bounded-asynchronous parameter servers for distributed ml," *arXiv preprint arXiv:1312.7869*, 2013.

[23] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[24] P. Sun, Y. Wen, T. N. B. D. Xiao *et al.*, "GraphH: High performance big graph analytics in small clusters," *arXiv preprint arXiv:1705.05595*, 2017.

[25] M. J. Anderson, N. Sundaram, N. Satish, M. M. A. Patwary, T. L. Willke, and P. Dubey, "Graphpad: Optimized graph primitives for parallel and distributed platforms," in *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 2016, pp. 313–322.

[26] M. Jain and C. Dovrolis, "End-to-end available bandwidth: measurement methodology, dynamics, and relation with tcp throughput," *IEEE/ACM Transactions on Networking (TON)*, vol. 11, no. 4, pp. 537–549, 2003.

[27] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," *The International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[28] Y. Gong, B. He, and J. Zhong, "Network performance aware mpi collective communication operations in the cloud," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 11, pp. 3079–3089, 2015.

[29] L. A. Barroso, J. Clidaras, and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines," *Synthesis lectures on computer architecture*, vol. 8, no. 3, pp. 1–154, 2013.

[30] O. Dekel, R. Gilad-Bachrach, O. Shamir, and L. Xiao, "Optimal distributed online prediction using mini-batches," *Journal of Machine Learning Research*, vol. 13, no. Jan, pp. 165–202, 2012.

[31] J. Bergstra, F. Bastien, O. Breuleux, P. Lamblin, R. Pascanu, O. Delalleau, G. Desjardins, D. Warde-Farley, I. Goodfellow, A. Bergeron *et al.*, "Theano: Deep learning on gpus with python," in *NIPS 2011, BigLearning Workshop, Granada, Spain*, 2011.

[32] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the ACM International Conference on Multimedia*. ACM, 2014, pp. 675–678.

[33] P. Sun, Y. Wen, T. N. B. Duong, and S. Yan, "Towards distributed machine learning in shared clusters: A dynamically-partitioned approach," *arXiv preprint arXiv:1704.06738*, 2017.

[34] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le *et al.*, "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems*, 2012, pp. 1223–1231.

[35] P. Moritz, R. Nishihara, I. Stoica, and M. I. Jordan, "SparkNet: Training deep networks in spark," *arXiv preprint arXiv:1511.06051*, 2015.

[36] F. N. Iandola, M. W. Moskewicz, K. Ashraf, and K. Keutzer, "FireCaffe: near-linear acceleration of deep neural network training on compute clusters," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2592–2600.

[37] P. Watcharapichat, V. L. Morales, R. C. Fernandez, and P. Pietzuch, "Ako: Decentralised deep learning with partial gradient exchange," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*. ACM, 2016, pp. 84–97.

[38] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Managed communication and consistency for fast data-parallel iterative analytics," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 2015, pp. 381–394.

[39] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "On parallelizability of stochastic gradient descent for speech dnns," in *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2014, pp. 235–239.

[40] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," *arXiv preprint arXiv:1502.02551*, 2015.

[41] Y. Zou, X. Jin, Y. Li, Z. Guo, E. Wang, and B. Xiao, "Mariana: Tencent deep learning platform and its applications," *Proceedings of the VLDB Endowment*, vol. 7, no. 13, pp. 1772–1777, 2014.