定时数据流

Timed Dataflow: Reducing Communication Overhead for Distributed Machine Learning Systems

许多分布式机器学习(ML)系统在处理大数据集时表现出较高的通信开销。我们的调查显示，流行的分布ML系统在网络通信上花费的时间可能比训练计算时间多一个数量级ML模型包含数百万个参数。这种高通信开销主要是由两个操作引起的:拉参数和推梯度。在本文中，我们提出了一种称为定时数据流(TDF)的方法，通过使用三种技术来减少网络流量来解决这个问题：一个定时参数存储系统，混合参数滤波器，混合梯度滤波器。特别的是，定时参数存储技术和混合参数过滤器使得服务器在pull操作中可以丢弃没有改变的参数，混合梯度过滤器允许服务器在push操作中有选择地降低梯度。因此,TDF可以显著减少网络流量和通信时间。

Peng Sun*†, Yonggang Wen†, Ta Nguyen Binh Duong†, Shengen Yan‡
*Interdisciplinary Graduate School, Nanyang Technological University, Singapore
†School of Computer Science and Engineering, Nanyang Technological University, Singapore
‡Sensetime Group Limited, Shatin, N.T., Hong Kong
{sunp0003, ygwen, donta}@ntu.edu.sg, yanshengen@sensetime.com

***Abstract*—Many distributed machine learning (ML) systems exhibit high communication overhead when dealing with big data sets. Our investigations showed that popular distributed ML systems could spend about an order of magnitude more time on network communication than computation to train ML models containing millions of *parameters*. Such high communication overhead is mainly caused by two operations: pulling *parameters* and pushing *gradients*. In this paper, we propose an approach called Timed Dataflow (TDF) to deal with this problem via reducing network traffic using three techniques: a timed parameter storage system, a hybrid parameter filter and a hybrid gradient filter. In particular, the timed parameter storage technique and the hybrid parameter filter enable servers to discard unchanged *parameters* during the pull operation, and the hybrid gradient filter allows servers to drop *gradients* selectively during the push operation. Therefore, TDF could reduce the network traffic and communication time significantly. Extensive performance evaluations in a real testbed showed that TDF could reduce up to 77% and 79% of network traffic for the pull and push operations, respectively. As a result, TDF could speed up model training by a factor of up to 4 without sacrificing much accuracy for some popular ML models, compared to systems not using TDF.**

***Keywords*-Machine Learning, Distributed System, Communication Overhead, Parameter Server**

I. Introduction

Machine learning (ML) systems automatically learn models from training data, and use them to predict values of future data. They are widely used in many domains, including image recognition, recommendation systems, text processing, etc. For example, logistic regression is widely used for advertisement prediction [1]. State-of-the-art deep learning systems can achieve high image classification accuracy [2]. Typically, a ML model consists of a large number of *parameters*. To minimize the prediction error, a ML system usually uses an iterative-convergent algorithm, such as stochastic gradient descent (SGD) [1], to obtain a set of optimal *parameters* from the training data. Existing single-node ML systems like Caffe [3] and Theano [4] could have acceptable execution time (ranging from several minutes to several hours) when dealing with a moderate amount of training data using iterative-convergent algorithms.

Due to the growth in training data size, distributed ML systems are getting popular. In industrial machine learning applications, the size of training data sets could be hundreds to thousands of terabytes [5]. For example, ImageNet [2] has hundreds of millions of records. Aforementioned single-node ML systems cannot cope with such large training data sets due to the limitation in storage and computation resource. Therefore, several distributed ML systems, for example MxNet [6] and Petuum [7], have been proposed to run the iterative-convergent algorithm based on the parameter server (PS) framework [1]. Such systems usually contains a group of *server* nodes and a group of *worker* nodes. The *server* nodes manage global *parameters* in a distributed shared memory system. During the training task, the *worker* nodes pull the latest *parameters* from *server* nodes, and push the calculated *gradients* to *server* nodes for the update operation. This is done iteratively to bring the model's *parameters* closer to the optimal values.

In existing PS-based distributed ML systems, a major challenge is the high communication overhead. During the training task, *parameters* and *gradients* are pulled and pushed frequently, generating a large amount of network traffic. Therefore, each worker spends a significant portion of time on the network communication. Our experiments show that the communication time could be an order of magnitude more than the computation time in many cases. In addition, network bottlenecks may also stall the computation if the synchronization latency is high [7]. Infiniband networking [8] is a recent approach to this problem. However, in most cloud computing platforms and clusters, only commodity Ethernet switches are available [9]. Therefore, network is becoming one of the main performance bottlenecks for PS-based distributed ML systems.

To alleviate the communication overhead in PS-based distributed ML systems, a number of approaches have been proposed. Stale Synchronous Parallel (SSP) [7] allows *worker* nodes to use stale *parameters*. As a result, SSP could overlap computation and communication, reducing the negative effect of communication overhead. Distributed Wait-free Backpropagation (DWBP) leverages the structure of convolutional neural networks (CNNs) and the chain rule of backpropagation (BP) algorithm to further overlap

1521-9097/16 $31.00 © 2016 IEEE
DOI 10.1109/ICPADS.2016.144

computation and communication when training CNNs. However, these approaches focus on reducing the synchronization latency without considering the network traffic problem. Thus, even with these approaches, the communication time could still be significant for some popular distributed ML applications, if the network bandwidth is limited.

这些方法关注于减少同步延迟，而不考虑网络流量问题。

In this paper, we propose Timed Dataflow (TDF), a type of network optimization for the PS framework, to tackle the high communication overhead. TDF aims to directly reduce network traffic and communication time by selectively dropping some *parameters* and *gradients* for the pull and push operations, respectively. More specifically, TDF employs three techniques: a timed parameter storage system, a hybrid parameter filter, and a hybrid gradient filter. The timed parameter storage system manages each *parameter* with a version number in *server* nodes. Therefore, *worker* nodes could reduce network traffic by not pulling unchanged *parameters*, which can be determined using their version numbers. The hybrid parameter/gradient filter could further reduce network traffic via dropping *parameters/gradients* selectively based on a set of predefined policies.

数据流

TDF的目标是通过选择性地降低拉、推操作的一些参数和梯度，直接减少网络流量和通信时间。

定时参数存储系统使用服务器节点中的版本号管理每个参数。因此，工作节点可以通过不拉取不变的参数来减少网络流量，这些参数可以使用版本号来确定。混合参数/梯度滤波器可以根据一组预定义的策略选择性地丢弃参数/梯度，从而进一步减少网络流量。

The primary contributions of this paper are as follows:

- We propose a new approach called TDF, which is a combination of three techniques to reduce the communication overhead for the PS framework;
- We design and develop a working implementation of TDF with a timed parameter storage system, a hybrid parameter filter and a hybrid gradient filter;
- We conduct extensive experiments using a working testbed with 7 physical servers and several popular ML models. The results show that TDF can reduce up to 77% and 79% of network traffic for the pull and push operation, respectively. As a result, TDF could speed up ML training by a factor of up to 4 without sacrificing much model's accuracy, compared to conventional PS-based systems without TDF.

问题识别：基于ps的分布式ML系统中的高通信开销

II. Problem Identification: High Communication Overhead in PS-based Distributed ML Systems

In this section, we first introduce PS and its two widely used synchronization techniques. Then we carry out a series of experiments to identify the high communication overhead.

A. The PS Framework

As shown in Figure 1, several popular distributed ML systems, such as MxNet, SparkNet, Adam, Factorbird and Petuum, adopt an approach referred to as *data parallelism*[1] from the PS framework [1]. A general PS-based distributed ML system contains a group of *server* nodes and a group of *worker* nodes. The *server* nodes manage global *parameters* in a distributed shared memory system. During the training

[1]*Model parallelism* is another approach, which is designed for big model problems. In this paper, we only consider *data parallelism*.

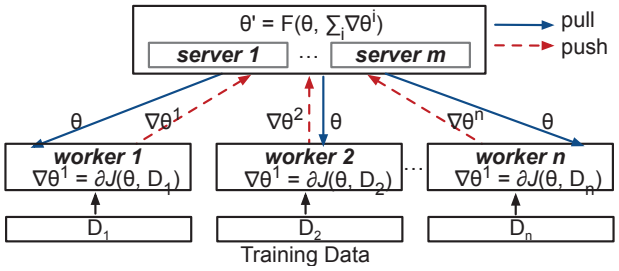


Figure 1. The PS framework: θ denotes *parameters*; $\nabla\theta$ denotes *gradients*; $J(\theta, D_n)$ is the predefined risk function for the ML model; $F(\theta, \sum_i^n \nabla\theta^i)$ is the *parameter* update function.

task, the *worker* nodes pull latest *parameters*, then calculate *gradients* using the allocated training data set, and finally push them to the *server* nodes for the update operation. This is done iteratively to bring the model's *parameters* closer to the optimal values. Two synchronization techniques are widely used in existing PS-based distributed ML systems: Bulk Synchronous Parallel (BSP) and Stale Synchronous Parallel (SSP) [7].

同步并行

1) BSP places a synchronization barrier at the end of each iteration. The *server* nodes need to collect *gradients* from all *workers* for the update operation at each iteration. As a result, at the beginning of each iteration, the *worker* nodes should wait for the completion of previous iteration, then they could pull the newest *parameters* for calculation.

BSP在每次迭代结束时都设置了同步障碍。(需要等待)

2) SSP allows *worker* nodes to use cached stale *parameters* as long as the cached version is within a staleness threshold. For example, at iteration t, a *worker* nodes can compute *gradients* directly without waiting for the newest *parameters*, if its cached parameters have been synchronized in at least one of the iterations from $t-s$ to $t-1$, where s is the staleness threshold. As a result, SSP could reduce the synchronization latency.

SSP允许工作节点使用缓存过时的参数，只要缓存版本在过时的阈值之内。

B. Communication Overhead in PS using AlexNet

网络通信是在PS框架中的一个严重的瓶颈

We argue that network communication is a serious performance bottleneck in the PS framework. To verify this performance bottleneck, we use lite-PS [1], a light-weight implementation of the PS framework, to implement a simple distributed ML system. The system contains two *worker* nodes and one *server* node. Each *worker* node contains NVIDIA Tesla K40 GPU, two Intel Xeon E5-2600 processors, and 128GB memory. The *server* node contains two Intel Xeon E5-2600 processors, and 128GB memory. The *worker* nodes and *server* node are connected with 1Gbps Ethernet. We integrate Caffe [3] into this system to train AlexNet [10], using ImageNet [2] as the training data set.

系统包含两个工作节点和一个服务器节点。

When using BSP, each *worker* node needs to pull 61.3M float *parameters* (approximately 245.2MB) and push 61.3M float *gradients* (approximately 245.2MB) on average in each iteration. According to our measurement, the pull and push operation would take around 12.8s on average. Compared to

当使用BSP的时候，每个工作节点需要推拉61.3M的参数和梯度每一次迭代。根据我们的测量，拉推操作平均需要12.8秒左右。

12.8s的通信时间，1.18s是计算时间

the computation time, which is around 1.18s per iteration, communication is the dominant component. When there are five *worker* nodes and one *server* node, the *server* node on average needs to receive 306.5M float *parameters* (approximately 1,226MB) and sends out 306.5M float *gradients* (approximately 1,226MB) in each iteration when training AlexNet. In this case, we can see that communication overhead could be the main performance bottleneck when using commodity Ethernet (i.e., 1 Gbps and 10 Gbps).

Though SSP and DWBP could reduce the negative effect of communication overhead, they cannot work well when the bandwidth is limited. SSP and DWBP focus on reducing the synchronization latency via overlapping computation and communication. In particular, SSP allows *worker* nodes to use stale *parameters*; and DWBP further overlaps computation and communication when training CNNs (for example, AlexNet). Such strategies work well when having enough bandwidth [9]. However, when using commodity Ethernet with limited bandwidth, SSP and DWBP could still have high communication overhead. We implemented SSP in our testbed to verify this. According to our measurement, when using SSP with a staleness threshold of 8, each pull operation takes 5.3s on average; each push operation takes 6.2s on average; and the computation takes around 1.18s per iteration. The experiments showed that SSP still has significant communication overhead, since it cannot overlap communication and computation efficiently due to the high volume of network traffic.

弄清楚SSP是怎么使用陈旧参数的?

III. Timed Dataflow: Reducing Network Traffic in PS-based Distributed ML Systems

整个论文中最重要的

In this section, we describe the architecture of Timed Dataflow (TDF), and show how it can reduce network traffic for the PS framework. Centralized to TDF is one strategy: dropping *parameters* and *gradients* selectively to reduce network traffic and communication time. To achieve this goal, TDF employs three techniques: a timed parameter storage system, a hybrid parameter filter, and a hybrid gradient filter. In short, the timed parameter storage system and the hybrid parameter filter enable PS *worker* nodes to drop *parameters* selectively in their pull operations. The hybrid gradient filter allows PS *worker* nodes to drop *gradients* selectively in their push operations.

有选择地丢弃参数和梯度，以减少网络流量和通信时间。

怎么来有选择的丢弃参数和梯度？

主要是利用定时参数存储系统和混合参数滤波器

A. TDF System Architecture

As shown in Figure 2, there are three key components in TDF, which is built over the PS framework:

一个定时参数存储系统，它使用两个子系统用版本号管理参数。服务器节点上的全局参数存储系统和工作节点上的本地参数缓存系统。

- A timed parameter storage system, which manages *parameters* with a version number using two subsystems: a global parameter storage system on the *server* nodes and a local parameter cache system on the *worker* nodes.

一个混合参数过滤器，它过滤掉服务器节点上的参数

- A hybrid parameter filter, which filters out *parameters* on the *server* nodes.

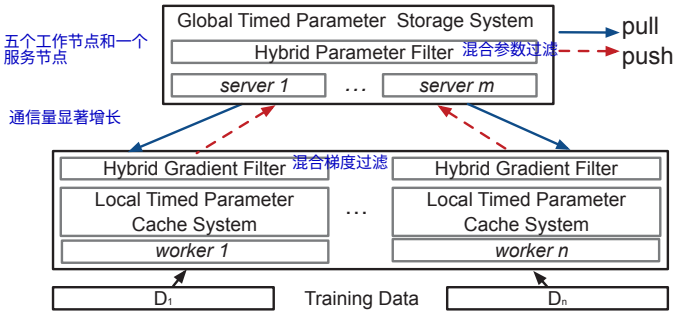


Figure 2. TDF's architecture with three key components: a timed parameter storage system, a hybrid parameter filter, and a hybrid gradient filter.

Algorithm 1 Training a ML model using TDF

1: **procedure** WORKER
2: Load the allocated training data sets.
3: Pull the latest *parameters* from TDF's timed parameter storage system on the *server* nodes.
4: Cache the pulled *parameters* in TDF's local timed parameter cache system.
5: Calculate *gradients*.
6: Push *gradients*, which are filtered out by TDF's hybrid gradient filter, to the *server* nodes.
7: **end procedure**

8: **procedure** SERVER
9: Calculate new values for *parameters* according to received *gradients*.
10: Update *parameters*, which are filtered out by TDF's hybrid parameter filter, to TDF's timed parameter storage system.
11: **end procedure**

从服务器节点上的TDF定时参数存储系统中提取最新的参数。

在TDF的本地时间参数缓存系统中缓存被拉参数。

通过TDF的混合梯度过滤器过滤掉的梯度推送到服务器节点。

根据接收到的梯度计算参数的新值。

将TDF混合参数滤波器滤波后的参数更新到TDF定时参数存储系统。

- A hybrid gradient filter, which filters out *gradients* the *worker* nodes.

TDF is a type of network optimization for the PS framework. Thus, TDF would not change the working flow of conventional PS-based distributed ML systems as shown in Algorithm 1.

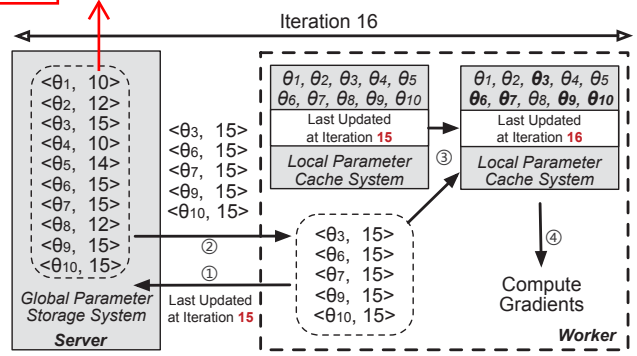
TDF vs. SSP. TDF and SSP are proposed to reduce communication overhead for the PS framework in different ways: SSP is a type of synchronization optimization; while TDF is a type of network optimization. More specifically, SSP lets workers use stale *parameters*, which may be different from the newest ones in the global parameter storage system. Thus, *worker* nodes would not wait for the slowest *worker* node at each iteration. Moreover, SSP could overlap computation and communication, since it enables *worker* nodes to perform the pull and push operations during the computation. As a result, SSP could reduce the communication time. However, as discussed in Section II, when the cluster has limited network bandwidth, the communication

TDF和SSP是以不同的方式来减少通信。

SSP是一种同步优化

SSP允许工人使用陈旧的参数，这可能与全局参数存储系统中的最新参数不同。而且工作节点不必等待最慢的工作节点。

这个方法可以借鉴。。。。 版本号 当一次迭代更新了一些节点的参数，则版本号会更新



在本例中，工作节点在迭代15时更新了缓存的参数。在迭代16中，在执行拉操作时，工作节点只需要拉版本号不小于15的参数，因为其他参数在前一次迭代中没有改变。被拉参数在工作节点的本地参数缓存系统中进行管理

Figure 3. Pull operation in TDF. In this example, a *worker* node has updated its cached *parameters* at iteration 15. At iteration 16, when performing the pull operations, the *worker* node only needs to pull *parameters* whose version numbers are not less than 15, since the others have not been changed in the previous iteration. The pulled *parameters* are managed in the *worker* node's local parameter cache system.

overhead is still significant. TDF is a complement approach, which directly reduces network traffic for the pull and push operations. TDF could work in conjunction with SSP to further reduce the communication overhead for distributed ML systems using the PS framework.

B. Timed Parameter Storage System 定时参数存储系统

TDF's timed parameter storage system enables *worker* nodes to distinguish changed/unchanged *parameters*. As a result, in the pull operation, *worker* nodes only need to pull changed *parameters*, to reduce network traffic. In TDF, the timed parameter storage system includes two subsystems:

全局
- **Global parameter storage system.** It manages newest *parameters*, which can be accessed by all *worker* nodes. In this subsystem, each *parameter* has a version number in the format of: $\langle \theta_j, v_j \rangle$, where v_j denotes the iteration number in which the jth *parameter* has been updated.

本地
- **Local parameter cache system.** It stores *parameters* pulled in previous iterations, and can be only accessed by the local *worker* node. This subsystem also keeps the latest iteration number when it pulls *parameters* from the global parameter storage system.

With the timed parameter storage system, TDF drops unchanged *parameters* during pull operations. Therefore, TDF could reduce network traffic as well as communication time. To achieve this goal, both pull and push operations would take the iteration number into account.

TDF's Pull Operation. PS *Worker* nodes avoid pulling unchanged *parameters* that are recognized by their version numbers. Each *worker* node maintains a local *parameter* cache system. Compared to the latest *parameters* stored in the global storage system, a portion of the local *parameters* might be unchanged. Thus, when pulling *parameters*, *worker* nodes only need to fetch changed *parameters*, and use them to update the cache system. To achieve this goal, we make *worker* nodes distinguish changed/unchanged *parameters* according to their version numbers. Specifically, each *worker* node keeps its latest iteration number w when it pulls *parameters*, and uses it to send out a pull request. When receiving a pull request, the global parameter storage system would look at all the *parameters*, and only return those whose version numbers are not less than w. The worker merges received *parameters* with its cached ones, and use them for gradient computation. As a result, TDF can reduce the network traffic for pull operations. Figure 3 gives an example with 10 *parameters*. In this example, at iteration 16, a worker needs to pull *parameters* whose version numbers are not less than 15. As a result, only 5 *parameters* are transferred via network for this pull operation.

具体来说，每个工作节点在提取参数时都会保留其最新的迭代号w，并使用它发送一个pull请求。

C. Hybrid Gradient Filter

TDF's hybrid gradient filter allows PS *worker* nodes to drop part of *gradients* selectively. Thus, *worker* nodes can reduce the network traffic and communication time when pushing *gradients* to PS *server* nodes. TDF employs three filtering polices: significantly modified, randomized, and limited numerical precision. TDF采用三种过滤策略:显著修改、随机化和有限的数值精度。

1) Significantly Modified Policy (G-SMP). With this policy, *worker* nodes drop *gradients* whose absolute values are less than a given threshold. It can be formulated as:

$$Drop\ \nabla\theta_j,\ \text{if}\ |\nabla\theta_j| < \frac{\phi_{gsmp}}{1 + d_{gsmp} \times \log t}, \qquad (1)$$

where $\nabla\theta_j$ is the *gradient* for the jth *parameter*, ϕ_{gsmp} is the initial threshold, t denotes the iteration number, and d_{gsmp} is a constant which can be chosen in advance. In the filter, $(1+d_{gsmp}\times\log t)$ works as a regularizer. In particular, it increases the threshold during the early stage of the training, dropping more *gradients* to reduce network traffic and communication time. At a later stage, the regularizer decreases the threshold, so that more tiny *gradients* could be used to obtain the optimal value faster.

2) Randomized Policy (G-RP). With this policy, workers randomly drop p_{grp} of *gradients*, where p_{grp} denotes the percentage of *gradients* to be dropped. For example, given a set of *gradients*, this policy would randomly drop 90% of *gradients* during the push operation, if $p_{grp} = 0.9$.

Table I
DEFINITIONS OF FLOATING POINT FORMATS.

Format		Total Bit-Width	Exponent Bit-Width	Mantissa Bit-Width
Double Precision:	*Float64*	64	11	52
Single Precision:	*Float32*	32	8	23
Half Precision:	*Float16*	16	5	10

3) Limited Numerical Precision Policy (G-LNP). With this policy, TDF uses floating point format with limited numerical precision to represent *gradients*, which can further reduce network traffic during the push operation. Formally,

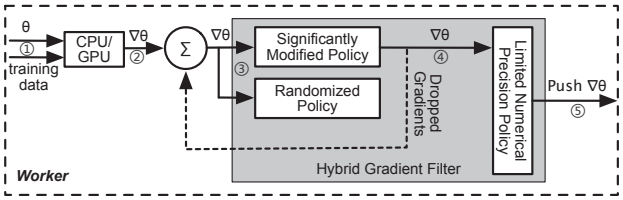


Figure 4. Push operation with TDF's hybrid gradient filter.

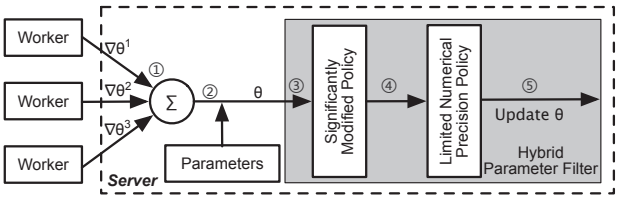


Figure 5. Update parameters with TDF's hybrid parameter filter.

floating point formats consist of a sign, an exponent (to give the floating point formats a wide range), and a mantissa (to give the floating point formats a good precision), as shown in Table I. Conventional systems usually use *Float32* to represent *parameters* and *gradients* for both computation and communication. It has been shown that low precision is sufficient not just for running trained ML models but also for training them [11] [12]. TDF adopts this policy to reduce network traffic, using *Float16* to represent *gradients* during the push operation.

Processing Dropped Gradients. TDF stores dropped *gradients* for aggregation in the next iteration. As shown in [13] and [7], all *gradients* work together to make *parameters* converge. Therefore, in TDF, the dropped *gradients* at iteration $t-1$ would be aggregated with the newly calculated *gradients* at iteration t using this formula:

$$\nabla\theta_j = \nabla\theta_j^{New} + \nabla\theta_j^{Dropped}, \tag{2}$$

where $\nabla\theta_j^{New}$ is the newly generated *gradient* in the current iteration. $\nabla\theta_j^{Dropped}$ is the dropped *gradients* in the previous iteration.

TDF's Push Operation. *Worker* nodes would not push *gradients* that are dropped by TDF's hybrid gradient filter. We design a new push algorithm using the hybrid gradient filter as shown in Figure 4. During the push operation, *worker* nodes should aggregate newly calculated *gradients* with dropped ones in previous iterations. Then, TDF's gradient filter would drop part of the *gradients* if they are selected by both *G-SMP* and *G-RP*. Finally, the remaining *gradients* would be pushed out after passing through *G-LNP*. Dropped *gradients* in this iteration are stored for aggregation at the *worker* nodes in the next iteration.

D. Hybrid Parameter Filter

TDF's hybrid parameter filter allows *server* nodes to drop some *parameters* selectively. Thus, *worker* nodes can reduce the network traffic and communication time when pulling *parameters* in later iterations. TDF employs two filtering polices in the hybrid parameter filter: significantly modified and limited numerical precision.

1) Significantly Modified Policy (P-SMP). With this policy, *server* nodes drop *parameters* whose changes are less than a given threshold. It can be formulated as:

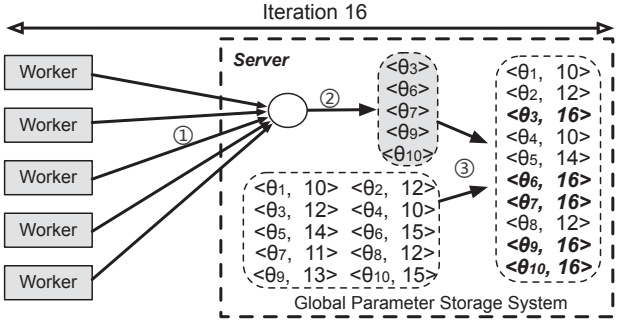


Figure 6. Update operation in TDF. In this example, only 5 *parameters* are changed at iteration 16.

$$\textit{Drop } \theta'_j, \text{ if } \left|\frac{\theta'_j - \theta_j}{\theta_j}\right| < \frac{\phi_{psmp}}{1 + d_{psmp} \times \log t}, \tag{3}$$

where θ_j is the original value of jth *parameter*, θ'_j is the new value of jth *parameter*, t denotes the iteration number, ϕ_{psmp} denotes the initial threshold, and d_{psmp} is a constant which can be chosen in advance. In the filter, $(1 + d_{psmp} \times \log t)$ works as a regularizer. In particular, it increases the threshold during the early stage of the training, dropping more *parameters* to reduce network traffic and communication time. At a later stage, the regularizer decreases the threshold, bringing the model's *parameters* closer to optimal values with tiny changing rates.

2) Limited Numerical Precision Policy (P-LNP). For this policy, we use *Float16* to store *parameters*, which is similar to *G-LNP*.

TDF's Update Operation. The *server* nodes would not update *parameters* that are dropped by TDF's hybrid parameter filter. We add TDF's hybrid parameter filter to the update operation as shown in Figure 5. During the update operation, the *server* nodes should first collect *gradients* from the *worker* nodes, and calculate new values of *parameters*. Then, TDF's parameter filter would drop part of the newly calculated *parameters*. The *server* nodes only update the remaining *parameters* to the global parameter storage system. For example, at iteration t, the aggregator would apply the aggregated gradient $\nabla\theta_j$ on the jth *parameter* θ_j, and obtain an updated *parameter*. In this case, the jth *parameter* in the global storage system would be updated to (θ'_j, t). Unchanged *parameters* at each iteration would keep

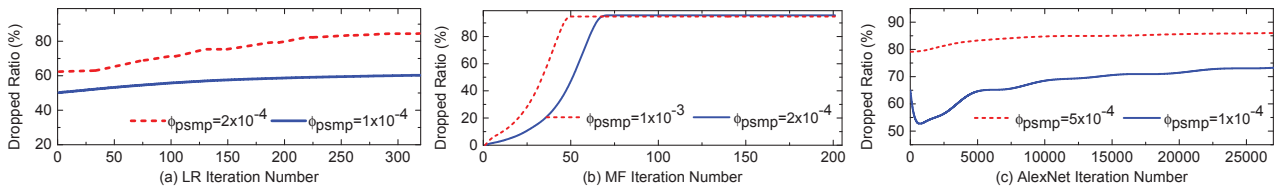


Figure 7. Percentage of dropped *parameters* when using TDF to train LR, MF, and AlexNet on BSP.

their original values and version numbers. Figure 6 gives an example with 10 *parameters*. This update operation could label changed and unchanged *parameters* between different iterations. Such labeling enables TDF to reduce network traffic and communication time for the pull operations in future iterations.

IV. Numerical Results and Analysis

In this section, we evaluate TDF's performance using an actual testbed. We first investigate TDF's impact on three popular ML models' quality, and then analyze TDF's performance improvement when working in conjunction with BSP and SSP. Due to space constraint, we only show some of the most representative results in this paper.

A. Experiments' Configuration

Hardware. We conduct experiments using a testbed with 7 physical Dell R720 servers. In particular, 5 servers act as *worker* nodes, each of which contains one NVIDIA Tesla K40 GPU accelerator, two Intel Xeon E5-2600 processors, 128GB memory, and 4TB HDD. Two remaining servers act as *server* nodes, each of which contains two Intel Xeon E5-2600 processors, 128GB memory, and 512GB SSD. All the servers are connected via 1Gbps Ethernet.

Software. We use Theano [4], Caffe [3] and Python to implement a PS-based distributed ML system with TDF's three key components. We use the system to train 3 ML models: Logistic Regression (LR), Matrix Factorization (MF) [14], and AlexNet [10] with the SGD solver. To verify the performance bottleneck, we run the system on BSP and SSPn, where n denotes the staleness threshold. It should be noted that all training data set are partitioned and equally distributed on the 5 *worker* nodes before the training task. Thus, there is no additional network overhead to fetch training data via the network. We use following data sets: *1)* We use a subset from Criteo [15] with 680 million records of ad clicks to train the LR model. At each iteration, a worker would roughly process 13.6 million records. *2)* We run MF on MovieLens [16] with roughly 22 million ratings from $33,000$ users on $240,000$ movies. At each iteration, a worker would roughly process 4 million ratings. *3)* We train AlexNet using a subnet of ImageNet [2], which contains 100k high resolution images with 1000 classes. At each iteration, a worker processes 256 images.

Performance Metrics. In the experiments, we collect the following data to verify the performance improvement: *1) Reduced network traffic in the pull operation.* We measure the percentage of dropped *parameters*, and the percentage of reduced network traffic in TDF's pull operation. *2) Reduced network traffic in the push operation.* We measure the percentage of dropped *gradients* and reduced network traffic in TDF's push operation. *3) Speedup ratio.* We measure the total execution time when using TDF to train ML models, and compare them to the non-TDF system.

Baseline. We disable TDF in our implemented distributed ML system, and use it as a baseline to demonstrate the performance gain of TDF.

B. Reduced Network Traffic in the Pull Operation

In this set of experiments, we measure reduced network traffic in the pull operation of TDF. We firstly collect the percentage of dropped *parameters*. In particular, we adjust the value of ϕ_{psmp} (the initial threshold of *P-SMP*), resulting in TDF dropping *parameter* with different thresholds. Then, we collect the percentage of dropped *parameters* at each iteration. It should be noted that we only show the data on BSP+TDF, since SSP+TDF updates *parameters* asynchronously, making it hard to be measured directly. Secondly, we measure the reduced network traffic generated by the pull operation.

Percentage of Dropped Parameters. As shown in Figure 7, TDF's hybrid parameter filter would drop a large portion of *parameters* for the three ML models. In particular, to train the LR model, TDF's hybrid parameter filter could drop up to 50% and 82% of *parameters* when using 1×10^{-4} and 2×10^{-4} as the value of ϕ_{psmp}, respectively. In MF, if we use the initial threshold 1×10^{-3} or 2×10^{-4}, TDF's hybrid parameter filter could drop up to 99% of *parameters* after the 80th iteration (according to our analysis, it is caused by the sparsity of the training data set, which could make a lot of *parameters* converge faster). The corresponding values for AlexNet are 65% and 80% when using 1×10^{-4} and 5×10^{-3} as the initial thresholds, respectively. From the results, we envisage that with a large enough ϕ_{psmp}, TDF's hybrid parameter filter could potentially help to reduce network traffic and communication time for the pull operations in distributed ML systems. In the following experiment, we use 1×10^{-4}, 2×10^{-4}, and 1×10^{-4} as the value of ϕ_{psmp} for LR, MF, and AlexNet, respectively.

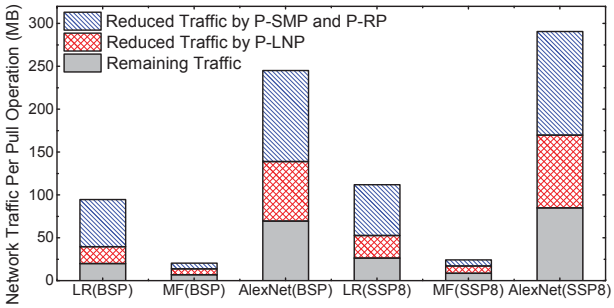


Figure 8. Average reduced network traffic per pull operation.

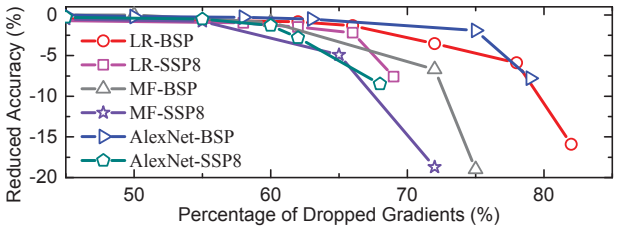


Figure 9. Percentage of dropped gradients in TDF, and its Relationship with prediction accuracy loss.

Percentage of Reduced Network Traffic for the Pull Operation. From Figure 8, we can observe that TDF's timed parameter storage system and hybrid parameter filter can significantly reduce the network traffic generated for the pull operation when training ML models. It should be noted that we use the values of ϕ_{psmp} selected in previous experiment. When working in conjunction with BSP, TDF could reduce 75%, 65%, and 77% of network traffic for the pull operation when training LR, MF, and AlexNet, respectively. The corresponding values when working with SSP8 are 74%, 62%, and 66%.

C. Reduced Network Traffic in the Push Operation

In this set of experiments, we measure the reduced network traffic in the push operation of TDF. We firstly measure the percentage of dropped *gradients*. In particular, we adjust the value of ϕ_{gsmp} (the initial threshold of *G-SMP*), making TDF drop *gradients* with different thresholds. For each ϕ_{gsmp}, we measure the percentage of dropped *gradients*, and the percentage of reduced prediction accuracy of the trained ML models due to gradient dropping. Secondly, we measure the reduced network traffic of the push operation.

Percentage of Dropped Gradients. As shown in Figure 9, for both BSP and SSP, a large portion of *gradients* can be dropped with acceptable impact on ML model quality. Specifically, when working in conjunction with BSP, TDF's hybrid gradient filter could drop roughly 58%, 55%, and 65% of *gradients* for LR, MF, and AlexNet respectively, with an accuracy reduction of around 0.5%. The corresponding ϕ_{gsmp} is 3×10^{-4}, 9.5×10^{-3}, and 2.7×10^{-2} for LR, MF, and AlexNet. We can also observe that when trying to drop more *gradients* with larger ϕ_{gsmp}, ML model quality would drop rapidly. For example, if we drop 75% of *gradients*, LR would have nearly 5% accuracy reduction. When working in conjunction with SSP8, TDF's hybrid gradient filter could drop roughly 55%, 50%, and 55% of *gradients* for LR, MF, and AlexNet, respectively, with an accuracy reduction of around 0.5%. The corresponding ϕ_{gsmp} is 5.5×10^{-4}, 5×10^{-3}, and 1.5×10^{-2} for LR, MF, and AlexNet, respectively. In the following experiments, we would use these values of ϕ_{gsmp}, since 0.5% prediction accuracy reduction is acceptable according to [17].

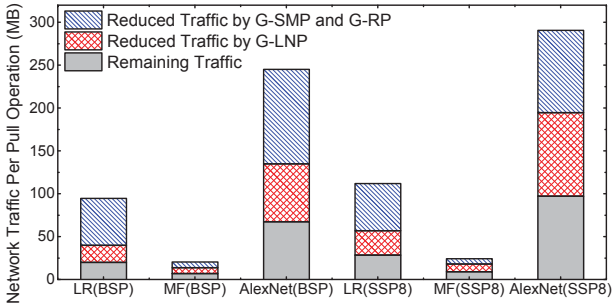


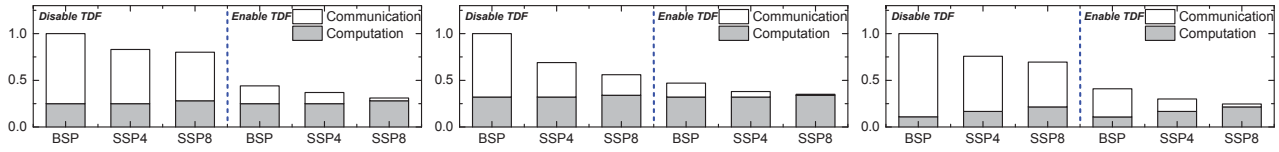
Figure 10. Average reduced network traffic per push operation.

Percentage of Reduced Network Traffic for the Push Operation. From Figure 10, we can observe that TDF's hybrid gradient filter can significantly reduce the network traffic for the push operation. It should be noted that we use the values of ϕ_{gsmp} selected in previous experiment. As shown in Figure 10(a), TDF's hybrid gradient filter could reduce roughly 79%, 66%, and 72% of network traffic when training LR, MF and AlexNet on BSP, respectively. The corresponding values for SSP8 are 76%, 64%, and 70%.

D. Speedup Ratio

In this set of experiments, we measure the speedup ratio when using TDF to train ML models. In particular, we enable TDF's three key components (which include hybrid gradient filter, hybrid parameter filter, and timed parameter storage system), and use it to train LR, MF, and AlexNet. We measure the total model training time, and compare it with the case without TDF. To verify the performance improvement, we run TDF on top of BSP, SSP4, and SSP8. We use the case of non-TDF system working in conjunction of BSP as the baseline in this experiment.

BSP's Speedup Ratio. We measure the total training time for TDF running on top of BSP. As shown in Figure 11 and Table II, TDF can speed up the training task considerably. Specifically, TDF can reduce the training time by a factor of 2.27, 2.13, and 2.45 for LR, MF and AlexNet, respectively. The performance improvement comes from the reduced network traffic and communication time. As shown in Figure 11, without TDF, the ratios of communication time



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

Figure 11. Communication and computation time ratio comparison between TDF and Non-TDF systems in terms of normalized training time.

to computation time for LR, MF and AlexNet training are $3.0 : 1$, $2.1 : 1$, and $8.1 : 1$, respectively. When TDF is enabled, the corresponding ratios become $0.76 : 1$, $0.47 : 1$, and $2.79 : 1$. From the results, we can clearly see that TDF could significantly reduce training time when working in conjunction with BSP.

Table II
TDF'S SPEEDUP RATIO

Models	Average Speedup Ratio		
	BSP	SSP4	SSP8
Logistic Regression	2.27±0.362	2.71±0.180	3.22±0.393
Matrix Factorization	2.13±0.727	2.63±0.531	2.85±0.511
AlexNet	2.45±0.461	3.34±0.277	4.06±0.637

SSP's Speedup Ratio. In this experiment, we measure the training time when using TDF in conjunction with SSP to train ML models. We observe that TDF can further reduce the training time. As shown in Table II, for AlexNet, TDF can reduce the training time by a factor of up to 4.06 when using SSP8. The corresponding values for LR and MF are 3.22 and 2.85, respectively. Figure 11 shows that the performance gain comes from reduced network traffic. In particular, the ratios of communication time to computation time for LR, MF and AlexNet training on TDF+SSP8 are $0.11 : 1$, $0.03 : 1$, and $0.15 : 1$, respectively. From the results, we can clearly see that TDF could significantly reduce training time when working in conjunction with SSP.

V. SUMMARY

In this paper, we propose *timed data-flow* (TDF) to speed up PS-based distributed ML systems with three techniques: a timed parameter storage system, and a hybrid gradient filter, and a hybrid parameter filter. Our experiments show that TDF can reduce up to 77% and 79% of network traffic for the pull and push operations when training popular ML models. Meanwhile, it can speedup the training task by a factor of up to 4 when working in conjunction with SSP. In the future, we plan to apply TDF to more applications.

REFERENCES

[1] 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.

[2] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, pp. 1–42, 2014.

[3] 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 MM*, 2014, pp. 675–678.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *NIPS*, 2012, pp. 1097–1105.

[11] M. Courbariaux, Y. Bengio, and J.-P. David, "Training deep neural networks with low precision multiplications," *arXiv preprint arXiv:1412.7024*, 2014.

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

[13] 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.

[14] S. Schelter, V. Satuluri, and R. Zadeh, "Factorbird-a parameter server approach to distributed matrix factorization," *arXiv preprint arXiv:1411.0602*, 2014.

[15] http://labs.criteo.com/downloads, 2015.

[16] http://grouplens.org/datasets/movielens, 2015.

[17] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *NIPS*, 2015, pp. 1135–1143.