A Proactive Data-Parallel Framework for Machine Learning

Guoyi Zhao, Tian Zhou and Lixin Gao
gzhao@umass.edu,tzhou@umass.edu,lgao@engin.umass.edu
Dept. of Electrical and Computer Engineering
University of Massachusetts Amherst

ABSTRACT

Data parallel frameworks become essential for training machine learning models. The classic Bulk Synchronous Parallel (BSP) model updates the model parameters through pre-defined synchronization barriers. However, when a worker computes significantly slower than other workers, waiting for the slow worker will lead to excessive waste of computing resources. In this paper, we propose a novel proactive data-parallel (PDP) framework. PDP enables the parameter server to initiate the update of the model parameter. That is, we can perform the update at any time without pre-defined update points. PDP not only initiates the update but also determines when to update. The global decision on the frequency of updates will accelerate the training. We further propose asynchronous PDP to reduce the idle time caused by synchronizing parameter updates. We theoretically prove the convergence property of asynchronous PDP. We implement a distributed PDP framework and evaluate PDP with several popular machine learning algorithms including Multilayer Perceptron, Convolutional Neural Network, K-means, and Gaussian Mixture Model. Our evaluation shows that PDP can achieve up to 20X speedup over the BSP model and scale to large clusters.

KEYWORDS

machine learning, asynchronous distributed computation, stragglers, gradient descent, expectation-maximization

ACM Reference Format:
Guoyi Zhao, Tian Zhou and Lixin Gao. 2021. A Proactive Data-Parallel Framework for Machine Learning. In *2021 IEEE/ACM 7th International Conference on Big Data Computing, Applications and Technologies (BDCAT '21) (BDCAT '21), December 6–9, 2021, Leicester, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3492324.3494167

1 INTRODUCTION

Machine learning (ML) has been widely applied in many fields. In computer vision, deep learning models such as convolutional neural networks can successfully detect objects [2], and recognize images [15, 18, 27]. In financial services, ML plays a key role in automatically detecting frauds and checking user identification [24, 34]. For natural language processing, ML becomes essential in writing articles and translating languages [6].

BDCAT '21, December 6–9, 2021, Leicester, United Kingdom

ACM ISBN 978-1-4503-9164-1/21/12…$15.00
https://doi.org/10.1145/3492324.3494167

With the explosion of data, it becomes challenging for a single machine to train machine learning models in a timely manner. It is essential to distribute the model training to a cluster of machines [10, 19, 28]. The most widely used data-parallel model is the Bulk Synchronous Parallel (BSP) model [30]. In BSP, the model parameters of ML algorithms are updated through synchronization. Each worker processes data points until it reaches a pre-defined synchronization point. When we adopt a parameter server architecture [19], the parameter server aggregates the processing results from workers and updates the model parameters. During the synchronization, every worker needs to wait until the model parameter has been updated and then continue the processing. The number of data points processed between synchronization points needs to be specified by the programmers before runtime. When a worker computes significantly slower than other workers (i.e., becomes a straggler), the pre-defined synchronization point will lead to excessive waste of computing resources. Stragglers can occur in many scenario [14], including heterogeneity and failures of hardware, unbalanced data distribution among tasks, using transient resources in the cloud [35]. For ML algorithms, the computation-intensive tasks amplify the waiting that can lead to significant straggler effects.

Distributed frameworks such as [19, 25, 26, 32] are proposed to support asynchronous data-parallel models. In these frameworks, each worker continues its data processing right after it contributes to the model parameters. Although asynchronous models remove the synchronization barrier, they suffer from another problem of *delayed updates*. That is, before a worker contributes to the model parameter, the model parameter may have already been updated by several other workers. Since the number of data points to be processed still needs to be specified before runtime, stragglers take a much longer time to finish a batch. So stragglers usually process data from an out-of-date model parameter, which will be less effective or even make a negative impact on the convergence speed [3, 21].

In this paper, we propose a novel data-parallel distributed framework for ML algorithms, a proactive data-parallel (PDP) framework. PDP reduces the impact of stragglers to accelerate the training. Instead of specifying a pre-defined synchronization point, PDP proactively decides when to update the model parameter at runtime. So the stragglers will not slow down other workers caused by waiting. The parameter server pulls the processing results from workers rather than waiting for workers to push the results. The global decision on the parameter server can provide workers with more up-to-date model parameters to accelerate the training.

PDP exploits the fact that an update of model parameters does not have to be performed after a pass of a fixed set of data points. The parameter server can not only pull from workers but also determine when to pull. The more proactively we pull, the more frequently the model parameter is updated. So that the workers can compute

updates from more up-to-date model parameters to potentially accelerate the training. However, the frequent pulling incurs overhead in interrupting workers and communicating intermediate results. Therefore, PDP determines the optimal time to pull at runtime according to different algorithms and datasets.

To further reduce the waiting time during synchronization, we propose an asynchronous proactive data-parallel (APDP) model to support asynchronous computations. APDP removes the synchronization barrier so that each worker can keep computing after responding to the pulling from the parameter server. Each worker uses the current model parameter to process data until the updated model parameters arrive. We theoretically prove that APDP guarantees the version differences of the model parameter are at most one among workers and the parameter server. So APDP still has the same convergence property as the BSP model.

Our major contributions are summarized as follows.

- We propose a new proactive data-parallel framework for iterative ML algorithms. PDP reduces the waiting time on workers especially when stragglers occur. PDP enables the parameter server to pull from workers so that there are no pre-defined synchronization points needed for workers. PDP not only pulls from workers but also determines the optimal time to pull. As a result, it will accelerate the training with an appropriate frequency of model parameter update.
- We propose APDP to further reduce the waiting time by removing synchronization barriers. Each worker keeps computing with its current model parameter instead of waiting for new model parameters. We theoretically prove that APDP can guarantee the version differences between a worker and the parameter server are at most one.
- We design and implement the distributed PDP and APDP. We evaluate PDP and APDP with several well-known ML algorithms, specifically Multilayer Perceptron, Convolutional Neural Network, K-means, and Gaussian Mixture Model on Google cloud clusters. We perform experiments on several straggler scenarios. The results show that we can achieve up to 20X speedup over synchronous models and 6X over asynchronous models.

The remainder of this paper is organized as follows. Section 2 describes how the ML algorithms are implemented under a data-parallel model. Section 3 introduces our PDP framework and Section 4 presents the design of the PDP system. We show the convergence property of the PDP framework in Section 5. Section 6 reports extensive evaluation results. Section 7 highlights the related works and Section 8 finally concludes this work.

2 MACHINE LEARNING ALGORITHMS

In this section, we show how machine learning algorithms are implemented under data-parallel models. We first describe the Gradient Descent (GD) algorithm that is widely used for model training. Then, we explain the well-known Expectation-Maximization (EM) algorithm in a distributed setting. In the end, we generalize how other algorithms can be executed in the data-parallel models.

2.1 Gradient Descent (GD) algorithm

Gradient descent is widely used in training machine learning models. It aims to learn the model parameters θ that minimizes an objective function known as the loss function. The loss function Q is usually computed as the summation of losses on all data points x_i in dataset X.

$$Q(\theta, X) = \sum_{x_i \in X} Q(\theta, x_i) \tag{1}$$

The GD algorithms iteratively re-estimate the model parameters θ to minimize the loss function. At iteration t, the model parameters are calculated through the gradient of the loss function on all the data in X.

$$\theta^{t+1} = \theta^t - \eta_t \sum_{x_i \in X} \nabla Q(\theta^t, x_i) \tag{2}$$

where η_t is the learning rate.

In a distributed environment, each worker j holds one partition of the dataset, $X_j \subset X$, and a copy of the model parameters θ_j. At iteration t, worker j computes the total gradient g_j^t, by summing up the gradients on all the data $x_i \in X_j$.

$$g_j^t = \sum_{x_i \in X_j} \nabla Q(\theta_j^t, x_i) \tag{3}$$

Then, the updates from all the P workers are aggregated together to re-estimate the model parameters as follows.

$$\theta^{t+1} = \theta^t - \eta_t \sum_{j=1}^{P} g_j^t \tag{4}$$

In a distributed environment, model parameters can be managed with a parameter server architecture [19]. The parameter server aggregates gradients from workers and accumulates them into model parameters. The aggregation is usually performed at a synchronization barrier. During the synchronization, workers send gradients g_j^t, referred to as *update*, to the parameter server and wait for new model parameters. The parameter server collects the gradients from workers, re-estimates the model parameters following Equation (4), and then broadcasts new model parameters to all workers.

For each update of the model parameters, processing a full batch of data points might not be necessary. A mini-batch GD is usually applied. Each worker j only processes a subset $B_j \subseteq X_j$ for an update. Therefore, we can re-estimate the model parameters more often, and the later mini-batch can make use of the more up-to-date model parameters to potentially improve the quality of gradients. Formally, at time t, the update g_j^t in Equation (3) is computed from a mini-batch B_j instead of the whole partition X_j.

$$g_j^t = \sum_{x_i \in B_j} \nabla Q(\theta_j^t, x_i) \tag{5}$$

In both full batch GD and mini-batch GD, batch size and consequently synchronization point are pre-defined before runtime. The fastest worker, which completes its mini-batch first, needs to wait for other workers to finish. Therefore, synchronization leads to overhead in a distributed environment.

2.2 Expectation-Maximization (EM) algorithm

We use the K-means algorithm as an example to introduce how to implement EM algorithms in a distributed environment through update aggregation. The K-means algorithm aims to partition N data points into K clusters. Formally, we represent each cluster with a centroid, which is the center of the cluster whose coordinators are the average of data points belonging to the cluster. K-means finds the assignments of all the data which minimizes the summation of the distance between each data point and its assigned cluster centroid. The model parameters for K-means consists of the K centroids $\theta = \{\theta_1, \cdots, \theta_K\}$ and the cluster assignments $Z = \{z_1, \cdots, z_N\}$. So the objective function $Q(\theta, Z, X)$ is

$$Q(\theta, Z, X) = \sum_{i=1}^{N} \sum_{k=1}^{K} \left[I(z_i, k) ||x_i - \theta_j||^2 \right] \tag{6}$$

where $I(a, b)$ is the indicator function which outputs 1 if and only if $a = b$, otherwise it outputs 0.

The K-means algorithm re-estimates Z and θ alternatively to minimize Q. First, given the current centroids θ, the algorithm assigns each data point to its nearest centroid as follows.

$$z_i = \underset{k \in \{1, \cdots, K\}}{\arg\min} ||x_i - \theta_j||^2 \tag{7}$$

Then, the algorithm re-estimates the centroids by averaging the coordinates of all data points assigned to the same cluster as the following equation.

$$\theta_k = \frac{\sum_{i=1}^{N} x_i I(z_i, k)}{\sum_{i=1}^{N} I(z_i, k)} \tag{8}$$

In a distributed environment, the assignments Z are computed on workers, while the centroids θ need to be aggregated based on the summation of $x_i I(z_i, k)$ and $I(z_i, k)$ from every worker. To efficiently re-estimate the centroids, we sum up the numerator and denominator in Equation (8) at each worker as *sufficient statistics*. So each worker j computes the sufficient statistics from a mini-batch $B_j \subseteq X_j$ for cluster k as

$$S_{j,k}^{t+1} = \sum_{x_i \in B_j} x_i I(z_{i,j}^t, k) \qquad C_{j,k}^{t+1} = \sum_{x_i \in B_j} I(z_{i,j}^t, k)$$

where $z_{i,j}^t$ represents the assignment of data x_i based on model parameter θ_j^t.

In the K-means, we can apply a delta change of the sufficient statistics to the total sufficient statistics to compute the centroids. So without synchronization, the new centroids can still be computed from the sufficient statistics of all the data. On worker j, the update is computed as the delta changes between two consecutive computations of sufficient statistics.

$$\Delta S_{j,k}^{t+1} = S_{j,k}^{t+1} - S_{j,k}^{t} \qquad \Delta C_{j,k}^{t+1} = C_{j,k}^{t+1} - C_{j,k}^{t}$$

Those updates can be aggregated through a centralized server or in a decentralized way as well. The model parameter θ_k^t can be computed from the update on worker j as

$$\theta_k^{t+1} = \frac{S_k^t + \Delta S_{j,k}^{t+1}}{C_k^t + \Delta C_{j,k}^{t+1}} \tag{9}$$

When using a centralized server, the new model parameter θ_k^{t+1} can be sent back to j or broadcast to all the workers. Or θ_k^{t+1} is directly apply to $\theta_{p,k}^{t+1}$ at any worker p that performs the re-estimation.

2.3 Other Machine Learning Algorithms

Many other machine learning algorithms can also be executed in a distributed fashion. Many ML algorithms try to learn a model through iteratively processing the given data. Each data point contributes updates towards the final model through update aggregation. So we can distribute the workload of processing data among workers while aggregating the updates to re-estimate the model. Similar to K-means, other algorithms, such as Gaussian Mixture Model (GMM), Nonnegative Matrix Factorization (NMF), and Latent Dirichlet Allocation (LDA), can also be implemented using a data-parallel model. As long as there are some sufficient statistics that can be aggregated, the algorithms can be executed under data-parallel models. In each iteration, we compute the sufficient statistics of all the data points based on current model parameters. Then we re-estimate the model parameters from sufficient statistics to maximize the likelihood. Specifically, each worker maintains a local copy of the model parameters and computes sufficient statistics (updates). Then the parameter server aggregates the updates from all the workers and re-estimates the model parameters through a synchronization barrier.

3 PROACTIVE DATA-PARALLEL FRAMEWORK

We propose a novel Proactive Data-Parallel (PDP) model to solve the problem of the slowdown caused by stragglers. In this section, we first give an overview of PDP in using a pulling mechanism to enable updating model parameters at any time. We then describe how PDP determines the optimal time to update model parameters. At last, we show the workflow of PDP and how we further reduce the idle time on workers with asynchronous computation.

3.1 Overview of PDP

PDP is designed to address the straggler problem to accelerate the training. A straggler is a worker that works significantly slower than other workers. To parallelize the computation, people typically distribute the workload to several workers. After every worker finishes its own workload, we perform a synchronization on the model parameter. However, the workload on each worker is pre-defined. When there is a straggler, all the workers have to wait for the straggler which leads to excessive waste of computing resources. Even we update model parameters in an asynchronous manner, a straggler takes a much longer time than other workers to contribute from an out-of-date model parameter. The pre-defined workload will make the updates from stragglers less effective or even make a negative impact on the convergence speed.

PDP determines the synchronization point based on how much workload has been done globally. Instead of assigning a workload to each worker beforehand, PDP monitors the global progress all the time. Therefore, the parameter server can use the global view of the progress to make a better synchronization point. To help the parameter server monitor the progress, workers periodically send

reports which contain the count of processed data points. We will discuss more details about the reports in Section 4.3.

At the synchronization point, the parameter server pulls from all the workers and aggregates them to the model parameter. Using the pulling mechanism, workers can keep processing the data points without waiting for other workers. Meanwhile, the stragglers can contribute in time. Since the pulling decision is the key in determining the efficiency of PDP, we will first describe the high-level idea.

3.2 Pulling Decision

A good pulling decision helps the PDP framework to accelerate the training. The more frequently the model parameter is updated, the workers can compute updates from more up-to-date model parameters to potentially accelerate the training. However, the frequent pulling incurs higher communication overhead and occupies more computing time. When the benefit of updating the model parameter covers the overhead, it is worthwhile to update. However, it is hard to quantify the benefit directly. Our goal is to accelerate the training, in other words, increase the convergence speed. We can estimate the relationship between the convergence speed and a pulling point to decide the optimal pulling time. Here, the *pulling point* k represents the number of data points we need to process from all the workers between two pullings.

We derive the relationship between convergence speed and different pulling points to find the optimal k. Generally, convergence speed is the ratio of the final loss function gain and the total training time which can only be learned after the training is completed. In order to find a good k at the beginning of the training process, we estimate the convergence speed after processing a number of data points. We measure the gain of loss function based on the model parameters after processing these data points. Therefore, the optimal pulling point k should have the highest gain as follows.

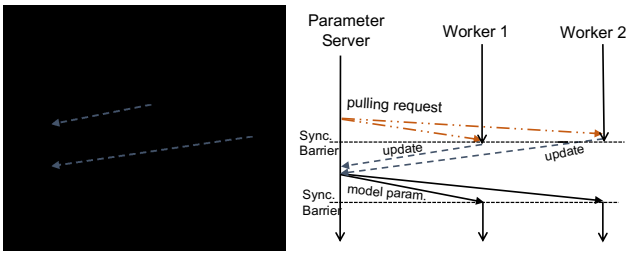
$$K^* = \arg\max_k \left(gain(k, N)\right) \tag{10}$$

Where the *gain function* $gain(k, N)$ presents the gain of loss function after processing N data points while updating the model parameter for processing every k data points. The gain function is affected not only by N and k, but also the starting model parameter, and the order of data points that are selected to process. We will show the details of how we select N and k to estimate $gain(k, N)$ in Section 4.2.

3.3 Synchronous PDP

The update of the model parameter can be performed through synchronization. In contrast with the widely-used bulk synchronous parallel (BSP) model, PDP does not need a pre-defined synchronization barrier. As shown in Figure 1a, the parameter server in BSP waits for all the workers to finish their computation and then synchronize the model parameter update. Therefore, a faster worker needs to wait for the slowest worker to finish, which leads to excessive waste of computing resources especially when there are stragglers. Also, it is hard to determine the number of data points that need to be processed for synchronization. That number usually varies for different algorithms, datasets, and infrastructure we use.

For synchronous PDP, the synchronization is initiated by the parameter server. As shown in Figure 1b, the parameter server first



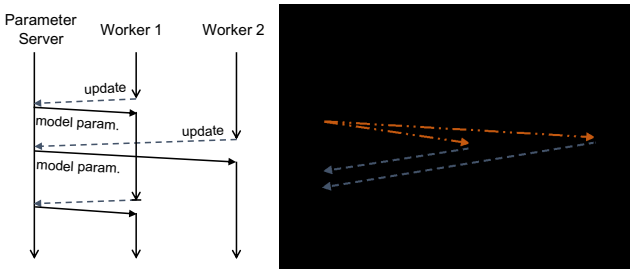
(a) BSP (b) PDP (Sync.)

Figure 1: Illustration of BSP (a) and PDP (Sync.) (b) on a cluster with one parameter server and two workers

broadcasts pull requests to all the workers in the synchronization. When workers reply to the parameter server, they pause the computation and wait for the new model parameter. So the convergence property in PDP can be still guaranteed as the same as BSP.

3.4 Asynchronous PDP

To make full use of the computation resources, we propose an asynchronous proactive data-parallel (APDP) model to further eliminate the idle time on workers during synchronization. After a worker sends updates, the computation has to pause in synchronous models. Any delay in synchronizing the updates, aggregating the updates to the model parameter, or sending the new model parameter will make idle time get longer. Since we still have the current model parameter, we can make use of the idle time to keep processing data. As shown in Figure 2b, after sending the updates, workers continue the computation with the current model parameters instead of waiting for the updated model parameters. Therefore, more data points can be processed instead of waiting.



(a) BAP (b) APDP

Figure 2: Illustration of BAP (a) and APDP (b) on a cluster with one parameter server and two workers

The asynchronous variant of PDP further reduces the waiting time. Comparing to the *basic asynchronous parallel* (BAP) model that is used in many frameworks [1, 9, 19, 26], APDP limits the delay updates. As shown in Figure 2a, BAP makes each worker pushes its update to the parameter server by its own choice. So there is no waiting for the stragglers. However, the BAP still requires a pre-defined number of data points to be processed for each update. When the straggling situation occurs, the slow worker takes a much longer time to contribute its updates that are based on an out-of-date

model parameter. It will make the updates less effective or even makes a negative impact.

In contrast to the BAP model, APDP guarantees the version differences between workers and the parameter server are at most one. Although the delay updates from the current model parameter still make the update inconsistent with the model parameter in the parameter server, we can bound the impact. We will show the theoretical analysis of the convergence property in Section 5.

4 DISTRIBUTED PDP SYSTEM

Now we represent our PDP distributed system. We first show an overview of the system. Then we explain the design of the parameter server and worker separately.

4.1 Overview of PDP system

Figure 3 shows an overview of the parameter server and workers, as well as the interactions between them. The parameter server manages the model parameters while the workers compute the updates from input data. For the parameter server, it controls the re-estimation of the model parameters. It also monitors the progress of workers to determine when to pull the updates from workers. The pulling determination module derives the pulling point based on the performance statistics of the parameter server and workers. For workers, each worker computes updates and reports its progress that is the number of data points processed from the last pulling.

Figure 3: Design of PDP framework

The parameter server and workers interact with each other through module communication. The parameter server broadcasts pulling requests to all the workers when the number of data points processed from all workers reaches K^*. Then the workers reply to the parameter server with its update. After accumulating updates to the model parameters, the parameter server broadcasts the new model parameters. The performance monitoring module helps the parameter server to determine when to pull. So the workers send the worker performance statistics along with the update.

4.2 Parameter Server Design

The parameter server maintains and re-estimates the model parameters, meanwhile, it pulls updates and determines when to pull. When the number of data points processed reaches K^* from the last pulling, the parameter server broadcasts the pulling requests. As described in Section 3.2, the gain function $gain(k, N)$ is a key factor for the efficiency of the PDP framework.

4.2.1 Estimate Gain Function. Based on Equation (10), the optimal k is estimated from the gain of loss function for processing N data points. Mathematically, the gain function $gain(k, N) = Q(\theta_k^N, X) - Q(\theta^0, X)$, where θ^0 and θ_k^N are the initial and final model parameters. To avoid additional overhead, we estimate $gain(k, N)$ during the training. We add a probing phase at the beginning of the training and keep the training progress. Generally, a large N, such as a full epoch, is more accurate to estimate $gain(k, N)$. However, the large N will increase the probing phase and delay the time to find the best K^*. To quickly estimate $gain(k, N)$, we only use a portion of a full batch to compute the $gain(k, N)$. We use a *probe ratio* pr towards the full batch to indicate the size of N. By default, we select $pr = 0.01$ for each k. We show that 0.5 is sufficient based on the experiments in Section 6.3. But the user can choose another pr to achieve a better estimation of $gain(k, N)$ or less probing time.

We adopt a multiplicative strategy to fast approach the optimal K^* instead of checking every possible k. From literature researches [5, 7, 20] and our experiments, we observed that the optimal mini-batch size is usually around 1% to 20% of the total data volume. Therefore, we start from the most possible optimal mini-batch $k_0 = 0.1 * M$ and gradually explore the optimal k from both sides. First, we compute $gain(k_0, pr * M)$ after processing one batch of $pr * M$ data points. Then, we compute $gain(k_1, pr * M)$ where k_1 is half of previous k_0. If the gains are smaller than k_0, we explore the other direction for larger k. For the case that $gain(k_0/2, pr * M)$ is better, we iteratively decrease the k_1 by half until a certain k has a smaller gain. Since each update incurs overhead in pulling and updating model parameters at the parameter server, the processing time of k data points cannot be shorter than this overhead. When we reach the minimum possible k, we stop probing. For the other case that $gain(k_0 * 2, pr * M)$ is better, we iteratively increase the k_2 by 2 until a certain k has a smaller gain or $pr * M$ reaches. Then the parameter server broadcasts the optimal K^* to every worker for count reports.

4.2.2 Pull Updates. When the number of data points reaches the optimal pulling point K^*, the parameter server broadcasts the pulling requests. The parameter server monitors the progress of workers by counting the reports sent by each worker after the last pulling. If each worker sends a counter of one for each data points it processed, the parameter server only needs to count them until K^* is reached. However, this will bring a huge overhead for both the parameter server and workers. It is not necessary to check every data points that have been processed either. Only the data points that are close to reaching K^* matter. Also, due to the communication overhead, the report itself takes time to reach the parameter server. Therefore, instead of counting the reports from workers for the accurate number of data points that have been processed, we estimate when the overall progress is about to reach K^*.

To estimate when to pull, we need to know the processing speed of each worker in runtime. But we want to limit the overhead of the count report. So the workers will send the count report when each worker reaches the $1/4$ of K^*/P where P is the number of workers. Then we use the time we received the count report between $1/4$, $1/2$,

and $3/4$ to estimate when to pull. For example, for worker i, we have the time to processed K^*/P as t_i. Then time we broadcast the pulling requests from the last pulling will be $T = 1/\sum_{i=1}^{P} t_i^{-1}$.

4.3 Worker Design

The main job of the workers is to compute updates and monitor the processed data points. The workers compute updates using their copy of the model parameters and respond to pulling requests and the arrived model parameters. After receiving a pulling request, the worker first finishes its atomic computation on the current data point. Then it sends out the accumulated updates since the last pulling. After processing the current data point, the worker pauses the computation and renews the model parameters. So the next data point can make use of the latest model parameters.

During the processing of data, the worker sends the count report to the parameter server. As described in Section 4.2.2, each worker knows the optimal K^* and sends the count report for every $K^*/P/4$ data points that have been processed. Therefore, there will be 3 count reports for each pulling which will not bring much overhead.

5 CONVERGENCE PROPERTY OF APDP MODEL

In this section, we use GD algorithms as an example to analyze the convergence property of APDP.

As described in Section 3, the gradient to re-estimate parameter θ^t is computed based on both of parameters θ^{t-1} and θ^t under APDP. Here we use A_t to represent all the data we process in mini-batch A_t at iteration t. In A_t, we divide the data into two parts B_t and C_t, which depends on the parameters we use. Suppose θ^{t-1} is applied to a subset B_t and θ^t is applied to C_t. The re-estimation of the model parameters using a learning rate η_t can be written as

$$\theta^{t+1} = \theta^t - \eta^t \left[\sum_{x_i \in B_t} \nabla Q(\theta^{t-1}, x_i) + \sum_{x_j \in C_t} \nabla Q(\theta^t, x_j) \right] \tag{11}$$

To bound the expected loss, we use the instantaneous regret $Q(\theta^t, X) - Q(\theta^*, X)$ to show the loss difference between θ^t and θ^*, where $\theta^* = \arg\min_\theta Q(\theta, X)$. By bounding the average regret from a sequence $\Theta = \{\theta^1, ..., \theta^T\}$ of parameters when T increase, we can say θ^T converges to θ^*.

In APDP, the gradients that are computed on model parameters θ^t come from two sets B_{t+1} and C_t. We denote the total set as $\tilde{A}_t = C_t + B_{t+1}$. So the instantaneous regret is computed as $Q(\theta^t, \tilde{A}_t) - Q(\theta^*, \tilde{A}_t)$, where $Q(\theta^t, \tilde{A}_t) = \sum_{x_i \in \tilde{A}_t} Q(\theta^t, x_i)$. Since the loss function Q is convex, the average regret we want to bound is as follows.

$$R[\Theta] := \frac{1}{T} \sum_{t=1}^{T} \left[Q(\theta^t, \tilde{A}_t) - Q(\theta^*, \tilde{A}_t) \right] \tag{12}$$

$$\leqslant \frac{1}{T} \sum_{t=1}^{T} \left\langle \nabla Q(\theta^t, \tilde{A}_t), \theta^t - \theta^* \right\rangle \tag{13}$$

$$\leqslant \frac{1}{T} \sum_{t=1}^{T} \left\langle \tilde{g}^t, \theta^t - \theta^* \right\rangle \tag{14}$$

Here we denote $\tilde{g}^t$ as the subdifferentials for Q, that is, $\tilde{g}^t = \sum_{x_i \in \tilde{A}_t} \nabla Q(\theta^t, x_i) = \sum_{i \in \tilde{A}_t} g_i^t$ where $g_i^t = \nabla Q(\theta^t, x_i)$.

To prove the regret bounds of GD in APDP, we can bound each instantaneous regret $\left\langle g^t, \theta^t - \theta^* \right\rangle$ at a given iteration t. So we first derive the following auxiliary lemma which bound the regret $\left\langle \sum_{j \in B_t} g_j^{t-1}, \theta^{t-1} - \theta^* \right\rangle + \left\langle \sum_{i \in C_t} g_i^t, \theta^t - \theta^* \right\rangle$. Then we show how the total regrets are bounded in theorem 5.1.

THEOREM 5.1. *Suppose the loss function Q is convex and Lipschitz continuous with a constant L, and $\max_{\theta, \theta' \in \Theta} D(\theta || \theta') \leqslant F^2$, given $\eta_t = \sigma/\sqrt{t}$ for some constant $\sigma > 0$ and maximum mini-batch size as M, the regret of GD in APDP is bounded by*

$$R[\Theta] \leqslant \frac{3\sigma M^2 L^2}{\sqrt{T}} + \frac{F^2}{\sigma\sqrt{T}}$$

and consequently for $\sigma = F/ML$ and we obtain the bound

$$R[\Theta] \leqslant \frac{4MFL}{\sqrt{T}}$$

The proof is shown in the document [1].

6 EXPERIMENTS

In this section, we evaluate our PDP framework to show its performance with several well-known ML algorithms. We use two EM algorithms, K-means and Gaussian Mixture Model (GMM), and two GD algorithms, Multi-Layer Perceptron (MLP) and Convolutional Neural Network (CNN) to show the efficiency of PDP and APDP under both homogeneous clusters and heterogeneous clusters. We also examine the effectiveness of the pull decisions in PDP. Finally, we test the scalability to show PDP can scale to large clusters.

6.1 Experiment Settings

We first describe our experiment settings including the algorithms, dataset, benchmark models, and testing environments. We build our PDP framework and implement the EM algorithms and GD algorithms using C++. We use OPEN MPI [13] to implement the distributed protocol for synchronous and asynchronous communications. Our code is publicly available [2].

6.1.1 Algorithms and Datasets. We test four representative ML applications to explore the performance of our PDP framework. We apply the K-means and Gaussian Mixture Model (GMM) [23] for the EM algorithms and Multilayer Perceptron (MLP) and Convolutional Neural Network (CNN) for the GD algorithms. We use publicly available datasets MASS [3] and HIGGS [4] from the high-energy physics field to evaluate the K-means and GMM algorithms. For the MLP and CNN, we test them on real-world dataset MNIST [5] and synthetic dataset. The summary of the datasets is shown in Table 1.

For K-means and GMM, we select $K = 50$ as the number of clusters. So the total parameter size is $K \times (d+1)$, d is the number of dimensions. Since K-means has been discussed in Section 2.2, now we explain the implementation of other algorithms.

[1] https://drive.google.com/file/d/141cPnxRKRxtDq_vWx2nh8aBMV9iw5Wju
[2] https://github.com/haku117/PDP
[3] http://archive.ics.uci.edu/ml/datasets/HEPMASS
[4] http://archive.ics.uci.edu/ml/datasets/HIGGS
[5] http://yann.lecun.com/exdb/mnist/

Table 1: Dataset Summary

Datasets	Points	Dimensions	Algorithms
MASS	7,000,000	27	EM: K-means/GMM
HIGGS	11,000,000	28	
MNIST	60,000	28x28 → 10	GD: MLP/CNN
Synthetic	1,000,000	1000x20 → 1	

GMM: The goal of GMM is to specify how likely a given data point X_i is generated from the j-th Gaussian distribution with the mean c_j and covariance matrix Σ_j, where $j \in 1, 2, ..., k$. $\theta_j = (c_j, \Sigma_j)$. The objective function is $f = \frac{1}{n}\sum_{i=1}^{n} \log(\sum_{j=1}^{k} w_j P(x_i|\theta_j))$ The statistics include three k-dimensions vectors, T, S and C which are computed as $T_j = \sum_{i=1}^{n} \gamma_{ij} X_i^2; S_j = \sum_{i=1}^{n} \gamma_{ij} X_i; C_j = \sum_{i=1}^{n} \gamma_{ij}$

The computation of update is to first compute the new value γ'_{ij}. Let $\delta = \gamma'_{ij} - \gamma_{ij}$, then summarize the statistics by: $T_j = T_j + \delta X_i^2$, $S_j = S_j + \delta X_i$, and $C_j = C_j + \delta$. Finally, the parameter server recomputes the model parameters as $w_j = C_j/n$, $c_j = S_j/C_j$, and $\Sigma_j = T_j/C_j - S_j^2/C_j^2$.

MLP: We set up a 3-layer Multi-Layer perceptron model for the MNIST dataset. There are $28 * 28 = 784$ input neurons and 10 output neurons. We set up 300 neurons in the hidden layer. Neurons are activated via the sigmoid function.

CNN: We design a CNN that has two convolutional layers with 10 and 20 2x2 kernels respectively. We use max-pooling and ReLU activation functions for each convolutional layer.

6.1.2 Benchmark models. We compare our PDP and APDP with synchronous model BSP and Asynchronous model BAP. Since the pulling determines the number of data points between two updates of model parameters, it is equivalent to a mini-batch size in BSP and BAP. Our PDP model is able to automatically determine the optimal global batch size, we do not need to set it up in advance. For BSP and BAP, we use 1% of input data as the mini-batch size following the literature researches [5, 7, 20]. Note that, since MLP and CNN are much more computation-intensive, we apply 0.2% as their mini-batch size.

6.1.3 Cluster Setting. We conduct our experiments on a Google Cloud cluster. We choose two types of instances to test the performance and scalability of our framework. One type is n1-standard8 with 8 CPU cores at 2.0-GHz and 7.5GB memory which is treated as a straggler. The other type is n2-highcpu8 with 8 CPU cores at 2.8-GHz and 7.5GB memory.

We organize instances into homogeneous clusters and heterogeneous clusters. We use 16 n2-highcpu8 instances to construct a homogeneous environment. We design two scenarios for the heterogeneous clusters with 16 instances as well. First, a *static heterogeneous cluster* contains two types of instances. We further simulate the different slowdowns by limiting the maximum CPU usage of slow workers. So we can show the performance with different scenarios. Second, in a *dynamic heterogeneous cluster*, the running speed of a worker may change over time. We simulate it by dynamically limiting the maximum CPU usage of a worker.

6.2 Efficiency of PDP framework

In this subsection, we evaluate the efficiency of our framework by comparing synchronous PDP and APDP with the BSP and BAP models. In the following, we evaluate the convergence speed on homogeneous clusters and heterogeneous clusters respectively.

6.2.1 Convergence Speed on Homogeneous Clusters. We first evaluate our PDP framework on homogeneous clusters, where every worker computes at the same speed. In Figure 4, we show the performance for different ML algorithms. For EM algorithms, we demonstrate the evaluation results for K-means algorithm on the MASS dataset in Figure 4a and GMM algorithm on the HIGGS dataset in Figure 4b. Since there is almost no additional waiting for workers on homogeneous clusters, PDP has a similar convergence speed as the BSP model. The value jump of the objective function in K-means around 25 seconds is when a full batch of data has been processed. In BAP model, it uses the assignments from out-of-date centroids to update the model parameter. So the convergence of the model parameter is slower. APDP can still outperform other models since it reduces the idle time during synchronization. APDP decreases the objective function faster from the beginning of the training and it achieves around 2X speedup towards BSP and 6X towards the BAP model.

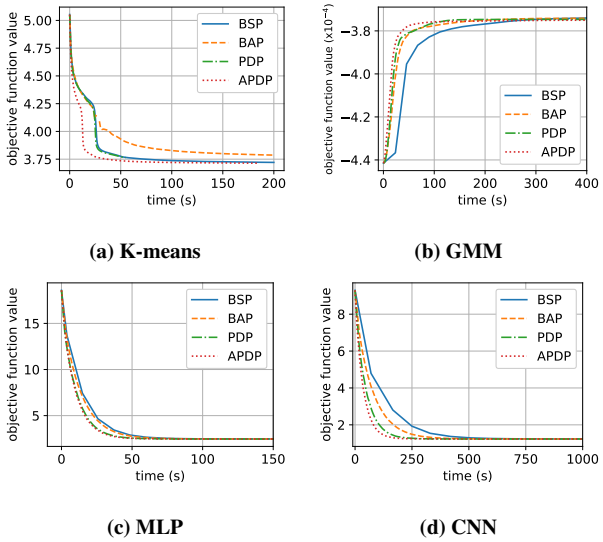


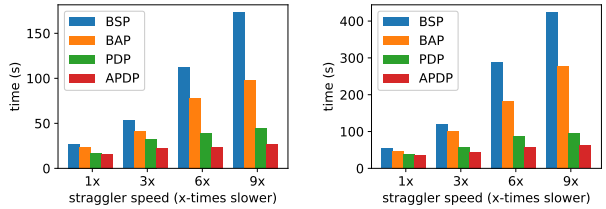
Figure 4: Runtime comparison among BSP, BAP and PDP

For GD algorithms like MLP in Figure 4c and CNN in Figure 4d, PDP and APDP achieve similar performance. Due to the stochastic nature of GD algorithms, these algorithms are not very sensitive to pulling decisions. As a result, although PDP and APDP can find a better update frequency, the speedup is not significant. Since they are more computation-intensive, the communication time is relatively short. So the waiting time during synchronization is not much to save for APDP. Generally, APDP is about 1.3X faster than BSP and BAP for MLP and 2.8X faster for CNN.

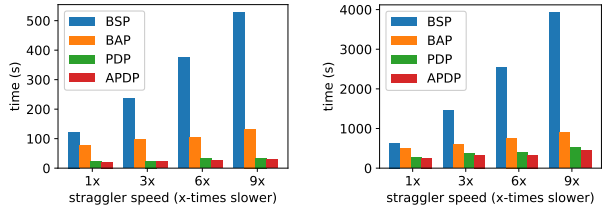
6.2.2 Convergence Speed on Heterogeneous Clusters. The pulling mechanism in the PDP framework makes each worker

contribute as much as they could. So that it can reduce the waiting time of fast workers when we train on heterogeneous clusters. In this subsection, we show the performance of our PDP framework in a variety of heterogeneous environments. We also use the HIGGS dataset for the K-means and GMM algorithms and the MNIST dataset for the MLP and CNN algorithms.

Static Heterogeneous Cluster: We first test the performance on a cluster with a worker that computes constantly slow. We set up a cluster with 16 workers, and one of them is a straggler which works slower than the other three. In order to show the impact of the straggler, we set up one straggler with different speeds, ranging from 1X slower to 9X slower. It is clear in the Figure 5 that no matter how slow the straggler is, the PDP and APDP framework outperforms the BSP and BAP model.



(a) K-means **(b) GMM**

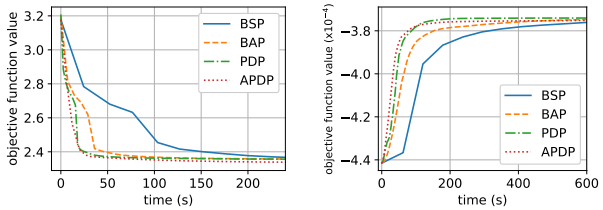


(c) MLP **(d) CNN**

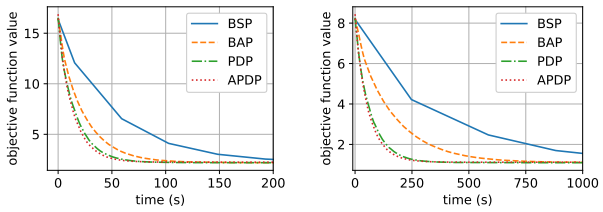
Figure 5: Convergence time comparison on static heterogeneous clusters. The X axis indicates the speed of the slowest worker

When the straggler is 9X slower than the normal workers, the PDP framework is about 4X faster than the BSP and 2.6X faster than the BAP model for the K-means algorithm. The delay is not as much because of the high ratio of communication time compared to the computation time. So APDP can further reduce the convergence time around 7X towards BSP and 5X towards the BAP model. A similar phenomenon is found in GMM. For GMM, PDP can speed up 3.2X and 2.4X towards BSP and BAP models. While APDP can speedup 7X and 4.6X towards BSP and BAP models For MLP and CNN, the computation time is dominant in training. So both PDP and APDP can reach a good speedup. Also, PDP can find a better update frequency when a straggler occurs. So APDP can achieve up to 20X faster than the BSP model and 6X faster than the BAP model.

Dynamic Heterogeneous Cluster: We also evaluate the performance when the processing speed of workers changes over time. This is a common phenomenon on private clusters and multi-tenant clouds where a limited amount of hardware resources is shared by multiple users. We designed our dynamic scenario as all workers periodically become slower or faster. The processing speed of the



(a) K-means **(b) GMM**

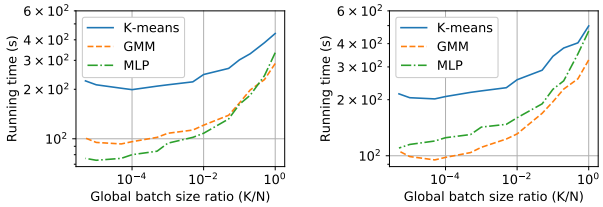


(c) MLP **(d) CNN**

Figure 6: Runtime comparison on dynamic heterogeneous clusters.

whole cluster gets changed noticeably. To simulate it, we randomly change the maximum CPU usage of all workers every 5 seconds.

We illustrate the training time in Figure 6. For the K-means and GMM algorithms, PDP and APDP decrease the objective function faster than the BSP and BAP models by about 6X and 2.1X. Especially when the algorithm almost converges, the PDP framework decreases the objective function faster. For the MLP and CNN algorithms, the PDP and APDP reach a similar convergence time. That is because they are more computation-intensive. The influence of the waiting time for synchronizing model parameters is relatively shorter than computation time. Generally, the speedup over the BSP model reaches up to 10X on dynamic heterogeneous clusters.



(a) Homogeneous clusters **(b) Heterogeneous clusters**

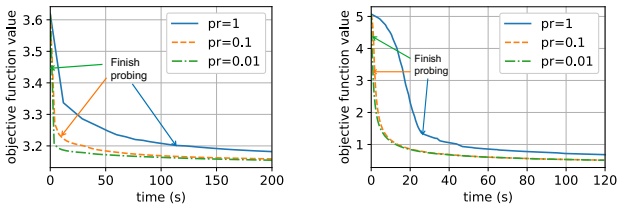
Figure 7: Impact of pulling

6.3 Impact of PDP Decisions

The pulling decision highly affects the performance of the PDP framework. Here, we show the impact of pulling point k in $gain(k, N)$ in Figure 7. On homogeneous clusters, K-means and GMM prefer a larger mini-batch size, which is a portion of 10^{-4} and 5×10^{-5} towards the full batch. For MLP, it prefers a smaller size which is 2×10^{-5} which is only around 20 data points per update. On heterogeneous clusters, the trend is similar to homogeneous clusters. In

PDP, we can find either the optimal or a little larger k which has the same scale with the convergence time increasing less than 3%.

Second, we examine the impact of pulling for N in $gain(k, N)$, which is based on the number of data to check for one update and the number of different k we need to probe. We select the different N in terms of the probe ratio pr. According to the probing ratio described in Section 4, a smaller pr indicates a smaller probing time but the probe result is not necessarily far away from the actual value. As shown in Figure 8, a small $pr = 0.01$ is able to help us determine optimal k. For K-means and MLP, we observe the same phenomenon.



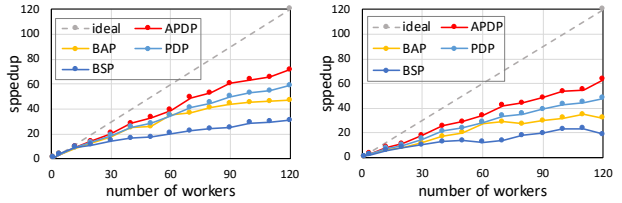
(a) K-means **(b) MLP**

Figure 8: Impact of probe ratio

6.4 Scalability

We further evaluate our PDP framework on the large-scale clusters to test its scalability. The scalability of different frameworks is tested on a homogeneous cluster and a static heterogeneous cluster with a delay up to 5 times slower than the normal run and there are 10% of the workers can be the straggler. We vary the total number of workers from 4 to 120.

We observe very good scalability results compared with BSP and BAP on both homogeneous and heterogeneous clusters in Figure 9. In BSP, the communication overhead grows dramatically. Using the same optimal batch size, which only incurs 1% overhead on communication on 4 workers, the overhead for 120 workers spends 99% of the training time. For BAP, it outperforms BSP since there is no synchronization cost. But the stale model parameters will still slow down the convergence speed. For our PDP, we automatically balance the ratio between data computation and update of model parameters by determining when to pull. So we still have a good speedup compared to BSP and BAP.



(a) Homogeneous cluster **(b) Heterogeneous cluster**

Figure 9: Scalability evaluation on PDP, BSP and BAP models

So with a large cluster of workers, the mini-batch size needs to get much larger than small clusters. In this scalability test, we select a mini-batch size as 5% of the total input data for BSP. It limits the synchronization overhead to less than 10% of the total training time. But the larger mini-batch size will make the time to use an up-to-date model parameter much later. So BSP will still be relatively slower.

7 RELATED WORKS

Many distributed frameworks [17, 22, 25, 32] adopt a centralized parameter server [19] for distributed implementation of machine learning algorithms. Machine learning algorithms infer models by refining model parameters. In a parameter server based distributed framework, workers compute the updates on the model parameters. The parameter server gathers the updates from workers and accumulates the updates to the model parameters.

Parameter server based distributed frameworks can update the model parameters in a synchronous fashion. That is, sending updates from workers and accumulating updates on the parameter server can be performed at a synchronization barrier. Each worker has to pause its computation of updates and wait for new model parameters from the parameter server to continue. A classic synchronous model is the bulk synchronous parallel (BSP) model [10, 28, 30]. However, synchronization may slow down the computation since workers have to wait for each other to reach the synchronization barrier. This is particularly true when there are stragglers that compute significantly slower than others [8, 14, 33] or using transient resources [35].

Several distributed systems have been proposed to reduce the synchronization overhead. For example, in K-sync SGD in [12], the parameter server waits for k workers to reach their synchronization points then updates the model parameters. The stale synchronous parallel (SSP) model [16] allows workers to skip synchronization barriers. However, each worker can skip at most a fixed number of steps (bounded staleness) before synchronization. SSP reduces the wait on synchronization but the faster workers still need to wait after the bounded staleness threshold is reached. FlexRR [14] combines the SSP model with the dynamic peer-to-peer reassignment of work among workers to further address the problem of stragglers. However, the overhead of migrating data can be huge.

FSP [31] and Sync-on-the-fly [35] propose a flexible synchronization barrier to reduce the impact of stragglers. Each worker can suspend the computation of updates when synchronizing with each other. Thus, synchronization barriers can be established at any time. Adaptive batch sizes are also proposed to optimize the mini-batch size during machine learning [4, 11]. However, even we determine the best mini-batch size, the computing resources are still wasted when all the workers pause during synchronization.

In contrast to the synchronous parallel model, the update of the model parameters from each worker can be sent to the parameter server in an asynchronous manner. Each worker sends the update to the parameter server at its own pace. Without waiting for other workers, the parameter server accumulates this update and sends the new model parameters back to the worker. Frameworks such as [19, 25, 26, 32] are used to support this asynchronous parallel model. MLNET [22] deploys a communication layer to implement asynchronous aggregation and ASYNC [29] build on top of Spark to support asynchronous computation. DistBelief [9] and TensorFlow [1] also support deep learning applications with asynchronous computation.

The asynchronous parallel model removes the synchronization overhead but usually suffers from the stale computation where workers use stale model parameters to compute updates. The stale computation of ML algorithms can slow down the convergence speed. To reduce the stale computation, K-async SGD in [12] waits for K workers to finish their mini-batch and then updates the model parameters. ASYNC [29] enables the workers and/or the parameter server to bookkeep (log) parameters to construct a dynamic dependence graph for the implementation with a partial broadcast of model parameters. These frameworks reduce the stale computation but still require a pre-defined model parameter update point. Stragglers can still slow down the computation.

Our PDP framework can update the model parameter at any time. Meanwhile, APDP is asynchronous. That is, workers do not wait for the update of the model parameters at the parameter server. Further, the framework determines when workers provide updates online. As a result, stragglers will not slow down the update of model parameters.

8 CONCLUSION

We propose a proactive data-parallel framework that enables the parameter server to initiate the update of model parameters at any time. PDP pulls from workers so that there are no pre-defined update points for workers and avoid workers waiting for each other. The parameter server cannot only pull from workers but also determine when to pull. The global decision on the parameter server can provide workers with more up-to-date model parameters to accelerate the training. We further propose asynchronous PDP to further reduce the idle time caused by synchronization. We theoretically prove the convergence property of APDP which shows the same result as PDP. We design and implement the PDP framework to determine the optimal time to pull from workers. Extensive experiments show that the PDP model consistently outperforms state-of-the-art solutions.

ACKNOWLEDGMENT

This work was supported in part by National Science Foundation Grants CNS-1815412 and CNS-1908536.

REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[2] A. Andreopoulos and J. K. Tsotsos. 50 years of object recognition: Directions forward. *Computer vision and image understanding*, 117(8):827–891, 2013.

[3] H. Avron, A. Druinsky, and A. Gupta. Revisiting asynchronous linear solvers: Provable convergence rate through randomization. *Journal of the ACM (JACM)*, 62(6):1–27, 2015.

[4] L. Balles, J. Romero, and P. Hennig. Coupling adaptive batch sizes with learning rates. *arXiv preprint arXiv:1612.05086*, 2016.

[5] L. Bottou, F. E. Curtis, and J. Nocedal. Optimization methods for large-scale machine learning. *Siam Review*, 60(2):223–311, 2018.

[6] T. Brants, A. C. Popat, P. Xu, F. J. Och, and J. Dean. Large language models in machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 858–867, 2007.

[7] R. H. Byrd, G. M. Chin, J. Nocedal, and Y. Wu. Sample size selection in optimization methods for machine learning. *Mathematical programming*, 134(1):127–155, 2012.

[8] J. Cipar, Q. Ho, J. K. Kim, S. Lee, G. R. Ganger, G. Gibson, K. Keeton, and E. Xing. Solving the straggler problem with bounded staleness. In *14th Workshop on Hot Topics in Operating Systems*, 2013.

[9] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, Ranzato, K. Yang, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[10] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[11] A. Devarakonda, M. Naumov, and M. Garland. Adabatch: Adaptive batch sizes for training deep neural networks. *arXiv preprint arXiv:1712.02029*, 2017.

[12] S. Dutta, G. Joshi, S. Ghosh, P. Dube, and P. Nagpurkar. Slow and stale gradients can win the race: Error-runtime trade-offs in distributed sgd. *arXiv preprint arXiv:1803.01113*, 2018.

[13] E. Gabriel, G. E. Fagg, et al. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 97–104. Springer, 2004.

[14] A. Harlap, H. Cui, E. P. Xing, and etc. Addressing the straggler problem for iterative convergent parallel ml. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 98–111. ACM, 2016.

[15] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[16] Q. Ho, J. Cipar, H. Cui, E. P. Xing, and etc. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in neural information processing systems*, pages 1223–1231, 2013.

[17] R. Jagerman and C. Eickhoff. Web-scale topic models in spark: An asynchronous parameter server. *arXiv preprint arXiv:1605.07422*, 2016.

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

[19] 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 *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 583–598, 2014.

[20] M. Li, T. Zhang, Y. Chen, and A. J. Smola. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD*, pages 661–670. ACM, 2014.

[21] X. Lian, Y. Huang, Y. Li, and J. Liu. Asynchronous parallel stochastic gradient for nonconvex optimization. In *Advances in Neural Information Processing Systems*, pages 2737–2745, 2015.

[22] 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.

[23] R. M. Neal and G. E. Hinton. A view of the em algorithm that justifies incremental, sparse, and other variants. In *Learning in graphical models*, pages 355–368. Springer, 1998.

[24] R. J. Orr and G. D. Abowd. The smart floor: A mechanism for natural user identification and tracking. In *CHI'00 extended abstracts on Human factors in computing systems*, pages 275–276. ACM, 2000.

[25] A. Qiao, A. Aghayev, W. Yu, H. Chen, Q. Ho, G. A. Gibson, and E. P. Xing. Litz: Elastic framework for high-performance distributed machine learning. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 631–644, 2018.

[26] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701, 2011.

[27] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv preprint arXiv:1312.6229*, 2013.

[28] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pages 1–10. Ieee, 2010.

[29] S. Soori, B. Can, M. Gurbuzbalaban, and M. M. Dehnavi. Async: A cloud engine with asynchrony and history for distributed machine learning. *arXiv preprint arXiv:1907.08526*, 2019.

[30] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[31] Z. Wang, L. Gao, Y. Gu, Y. Bao, and G. Yu. Fsp: towards flexible synchronous parallel framework for expectation-maximization based algorithms on cloud. In *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 2017.

[32] 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. *IEEE Transactions on Big Data*, 1(2):49–67, 2015.

[33] J. Yin, Y. Zhang, and L. Gao. Accelerating expectation-maximization algorithms with frequent updates. In *2012 IEEE International Conference on Cluster Computing*, pages 275–283. IEEE, 2012.

[34] R. Zafarani and H. Liu. User identification across social media, Jan. 10 2017. US Patent 9,544,381.

[35] G. Zhao, L. Gao, and D. Irwin. Sync-on-the-fly: A parallel framework for gradient descent algorithms on transient resources. In *IEEE International Conference on Big Data*, pages 392–397. IEEE, 2018.