Dynamic Backup Workers for Parallel Machine Learning

Chuan Xu Giovanni Neglia Nicola Sebastianelli
Inria, Université Côte d'Azur, Sophia Antipolis, France,
firstname.familyname@inria.fr

***Abstract*—The most popular framework for parallel training of machine learning models is the (synchronous) parameter server (PS). This paradigm consists of n workers, which iteratively compute updates of the model parameters, and a stateful PS, which waits and aggregates all updates to generate a new estimate of model parameters and sends it back to the workers for a new iteration. Transient computation slowdowns or transmission delays can intolerably lengthen the time of each iteration. An efficient way to mitigate this problem is to let the PS wait only for the fastest $n-b$ updates, before generating the new parameters. The slowest b workers are called *backup workers*. The optimal number b of backup workers depends on the cluster configuration and workload, but also (as we show in this paper) on the hyper-parameters of the learning algorithm and the current stage of the training. We propose DBW, an algorithm that dynamically decides the number of backup workers during the training process to maximize the convergence speed at each iteration. Our experiments show that DBW 1) removes the necessity to tune b by preliminary time-consuming experiments, and 2) makes the training up to a factor 3 faster than the optimal static configuration.**

I. Introduction

In 2014, Google's Sybil machine learning (ML) platform was already processing hundreds of terabytes through thousands of cores to train models with hundreds of billions of parameters [1]. At this scale, no single machine can solve these problems in a timely manner, and, as time goes on, the need for efficient parallel solutions becomes even more urgent. Currently, the operation of ML parallel systems requires a number of ad-hoc choices and time-consuming tuning through trial and error, e.g. to decide how to distribute ML programs over a cluster or how to bridge ML computation with inter-machine communication. For this reason, significant research effort (also from the networking community [2], [3], [4], [5]) is devoted to design adaptive algorithms for a more effective use of computing resources for ML training.

Currently, the most popular template for parallel ML training is the parameter server (PS) framework [6]. This paradigm consists of workers, that perform the bulk of the computation, and a stateful parameter server that maintains the current version of the model parameters. Workers use locally available versions of the model to compute gradients which are then aggregated by the PS and combined with its current state to produce a new estimate of the optimal parameter vector. If the PS waits for all workers before updating the parameter vector (synchronous operation), *stragglers*, i.e. slow tasks, can significantly reduce computation speed in a multi-machine setting [7], [8], [9]. Transient slowdowns are common in computing systems (especially in shared ones) and have many causes, such as resource contention, background OS activities, garbage collection, and (for ML tasks) stopping criteria calculations. Alternatively, the PS can operate asynchronously, updating the parameter vector as soon as it receives the result of a single worker. While this approach increases system throughput (parameter updates per time unit), some workers may operate on stale versions of the parameter vector slowing and, in some cases, even preventing convergence to the optimal model [10]. A simple solution that does not jeopardize convergence, while mitigating the effect of stragglers, is to rely on backup workers [11]: instead of waiting for the updates from all workers (say it n), the PS waits for the fastest k out of n updates to proceed to the next iteration. The remaining $b \triangleq n-k$ workers are called backup workers. Experiments on Google cluster with $n=100$ workers show that a few backup workers (4–6) can reduce the training time by 30% in comparison to the synchronous PS and by 20% in comparison to the asynchronous PS [11].

The number of backup workers b has a double effect on the convergence speed. The larger b is, the faster each iteration is, because the PS needs to wait less inputs from the workers. At the same time, the PS aggregates less information, so the model update is noisier and more iterations are required to converge.

Currently, the number of backup workers is configured manually through some preliminary experiments, before the actual training process starts. However, the optimal static setting is highly sensitive to the cluster configuration (e.g. GPU performances and their connectivity), as well as its instantaneous workload, that may be unknown to the users (specially in a virtualized cloud setting) and may change as new jobs arrive/depart from the cluster. Moreover, in this paper we show that the optimal number of backup workers 1) is also affected by the choice of hyper-parameters like the batch size, and 2) changes during the training itself(!) as the loss function approaches a (local) minimum. Therefore, the static configuration of backup workers does not only require time-consuming experiments, but is particularly inefficient and fragile.

In this paper we propose the algorithm DBW (for Dynamic Backup Workers) that dynamically adapts the number of backup workers during the training process without prior knowledge about the cluster or the optimization problem. Our algorithm identifies the sweet spot between the two contrasting effects of b (reducing the duration of an iteration and increasing the number of iterations for convergence), by maximizing at each iteration the decrease of the loss function *per time unit*.

The paper is organized as follows. Sect. II provides relevant background and introduces the notation. Sect. III illustrates the different components of our algorithm DBW with their respective preliminary assessments. DBW is then evaluated on ML problems in Sect. IV. The results show that DBW is robust to different cluster environments, and different hyper-parameters' settings. DBW does not only remove the necessity to configure an additional parameter (b) through costly experiments, but also reduce the training time by a factor as large as 3 in comparison to the best static configuration. Sect. V concludes the paper and discusses future research directions. Our code is available online [12].

II. Background and notation

Given a dataset $\mathbb{X} = \{x_l, l = 1, \dots S\}$, the training of ML models usually requires to find a parameter vector $\boldsymbol{w} \in \mathbb{R}^d$ minimizing a loss function:

$$\underset{\boldsymbol{w}\in\mathbb{R}^d}{\text{minimize}} \quad F(\boldsymbol{w}) = \frac{1}{S}\sum_{l=1}^{S} f(x_l, \boldsymbol{w}), \tag{1}$$

where $f(x_l, \boldsymbol{w})$ is the loss of the model $\boldsymbol{w}$ on the datapoint x_l. For example, in supervised learning, each point of the dataset is a pair $x_l = (\chi_l, y_l)$, consisting of an input object χ_l and a desired output value y_l. In the standard linear regression method $\chi_l \in \mathbb{R}^d$, $y_l \in \mathbb{R}$, the input-output function is a linear one ($\hat{y}_l = \chi_l^\intercal \boldsymbol{w}$) and the loss function is the mean squared error $(\chi_l^\intercal \boldsymbol{w} - y_l)^2$. More complex models like neural networks look for an input-output mapping in a much larger and more flexible family of functions, but they are trained solving an optimization problem like (1).

The standard way to solve Problem (1) is to use an iterative gradient method. Let n be the number of workers (e.g. GPUs) available. In a synchronous setting without backup workers, at each iteration t the PS sends the current estimate of the parameter vector $\boldsymbol{w}_t$ to all the workers. Each worker computes then a stochastic gradient on a random mini-batch of size B ($\leq S$) drawn from its local dataset. We assume each worker has access to the complete dataset $\mathbb{X}$ as it is reasonable in the cluster setting that we consider. Each worker sends the stochastic gradient back to the PS. We denote by $\boldsymbol{g}_{i,t}$ the i-th worker gradient received by the PS at iteration t, i.e.

$$\boldsymbol{g}_{i,t} = \frac{1}{B}\sum_{x\in\mathbb{B}_i} \nabla f(x, \boldsymbol{w}_t), \tag{2}$$

and $\mathbb{B}_i \subseteq \mathbb{X}$ is the random minibatch of size B on which the gradient has been computed. Once n gradients are received, the PS computes the average gradient

$$\boldsymbol{g}_t = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{g}_{i,t},$$

and updates the parameter vector as follows:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta \boldsymbol{g}_t, \tag{3}$$

where $\eta > 0$ is called the learning rate.

When b backup workers are used [11], the PS only waits for the first $k = n - b$ gradients and then evaluates the average gradient as

$$\boldsymbol{g}_t = \frac{1}{k}\sum_{i=1}^{k} \boldsymbol{g}_{i,t}. \tag{4}$$

In our dynamic algorithm (Sect. III), the value of k is no longer static but changes in an adaptive manner from one iteration to the other, ensuring faster convergence speed. We denote by k_t the number of gradients of $\boldsymbol{w}_t$ the PS needs to wait for at iteration t, and by $T_{i,t}$ the time interval between the update of the parameter vector $\boldsymbol{w}_t$ at the PS and the reception of the i-th gradient $\boldsymbol{g}_{i,t}$.

The general backup-workers scheme can be implemented in different ways. The updated parameter vector could be either pulled by idle workers (as in Google's TensorFlow framework) or pushed by PS. When PS pushes the parameter vector, PS could either force workers to *interrupt* their ongoing gradient computation (PsI) or *wait* for them to complete it (PsW). Our algorithm works with all variants listed above, with minor adaptations. We have implemented and tested it both with PsI and PsW in the PyTorch framework [13]. Results are similar, therefore, in what follows, we refer only to the PsW.

To the best of our knowledge, the only other work proposing to dynamically adapt the number of backup workers is [14]. The authors consider a PsI approach. The PS uses a deep neural network to predict the time $T_{k,t}$ needed to collect $k = 1, 2, \dots n$ new gradients. It then greedily chooses k_t as the value that maximizes $k/T_{k,t}$. The neural network for time series forecasting needs itself to be trained in advance for each cluster and each ML model to be learned. No result is provided in [14] about the duration of this additional training phase or its sensitivity to changes in the cluster and ML models. Our algorithm DBW also selects k_t to maximize a similar ratio, but 1) replaces the numerator by the expected decrease of the loss function, 2) uses a simple estimator for $T_{k,t}$, that does not require any preliminary training. Moreover, results in [14] do not show a clear advantage of the proposed mechanism in comparison to the static setting suggested in [11] (see e.g. [14, Fig. 4]). Our experiments in Sect. IV confirm that indeed considering a gain proportional to k as in [14] is too simplistic (and leads to worse results than DBW).

Our approach to estimate the loss decrease as a function of k is inspired by the work [15] which evaluates the loss decrease as a function of the batch size. In fact, aggregating k gradients, each computed on a mini-batch of B samples, is almost equivalent to compute a single gradient on a mini-batch of kB samples.

While our algorithm adapts the number of backup workers b given an available pool of n workers, the authors of [16] propose a reinforcement learning algorithm to adapt n in order to minimize the training time under a budget constraint. This algorithm and DBW are then complementary: once selected n with the approach in [16], DBW can be applied to tune the number of backup workers.

III. Dynamic backup workers

The rationale behind our algorithm DBW is to adaptively select k_t in order to maximize $\frac{F(\boldsymbol{w}_t)-F(\boldsymbol{w}_{t+1})}{T_{k,t}}$, i.e., to greedily maximize the decrease of the empirical loss per time unit. We decide k_t just after the update of $\boldsymbol{w}_t$. In the following subsections, we detail how both numerator and denominator can be estimated, and how they depend on k. The notation is listed in Table I.

n, t	number of workers, iteration t
$F, \boldsymbol{w}_t$	(global) loss function to minimize, parameter vector
L, B, η	Lipschitz smoothness constant, batch size, learning rate
$\boldsymbol{g}_{i,t}$	i^{th} stochastic gradient PS receives at iter. t
$\mathbb{V}(\boldsymbol{g}_{i,t})$	variance of $\boldsymbol{g}_{i,t}$
k_t	number of stochastic gradients PS waits for at iter. t
$\boldsymbol{g}_t$	average gradient at iter. t
$\mathcal{G}_{k,t}$	gain (expected loss decrease) if PS receives k gradients
$T_{k,t}$	time between $\boldsymbol{w}_t$ update and $\boldsymbol{g}_{k,t}$ reception at PS
$\mathtt{t}_{h,i,t}$	time between $\boldsymbol{w}_t$ update and $\boldsymbol{g}_{k,t}$ reception at PS when PS has waited for h gradients at iter. $t-1$
$\mathcal{T}_{h,k}$	random variable from which $\mathtt{t}_{h,i,t}$ values are assumed to be sampled
$\mathbb{T}_{h,k,t}$	set of $\mathtt{t}_{h,k,t'}$ samples available up to iter. t

TABLE I: Notation

A. Empirical Loss Decrease

We assume that the empirical loss function $F(\boldsymbol{w})$ is L-smooth, i.e., it exists a constant L such that

$$\|\nabla F(\boldsymbol{w}') - \nabla F(\boldsymbol{w}'')\| \leq L\|\boldsymbol{w}' - \boldsymbol{w}''\|, \forall \boldsymbol{w}', \boldsymbol{w}''. \quad (5)$$

Smoothness is a standard assumption in convergence results of gradient methods (see for example [17], [18]). In our experiments we show DBW reduces the convergence time also when the loss is not a smooth function. From (5) and (3) it follows (see [18, Sect. 4.1] for a proof):

$$\begin{aligned} \Delta F_t &\triangleq F(\boldsymbol{w}_t) - F(\boldsymbol{w}_{t+1}) \\ &\geq \eta \nabla F(\boldsymbol{w}_t)^\intercal \boldsymbol{g}_t - \frac{L\eta^2}{2}\|\boldsymbol{g}_t\|^2. \end{aligned} \quad (6)$$

In order to select k_t, DBW uses this lower bound as a proxy for the loss decrease. We note, however, that $\boldsymbol{g}_t$ depends on the value of k_t (see (4)) and the random mini-batches drawn at the workers. So at the moment to decide for k_t, $\boldsymbol{g}_t$ is a random variable. We consider then the expected value (over the possible choices for the mini-batches) of the right-hand side of (6). We call it the *gain* and denote by $\mathcal{G}_{k,t}$, i.e.:

$$\mathcal{G}_{k,t} \triangleq \mathbb{E}\left[\eta \nabla F(\boldsymbol{w}_t)^\intercal \boldsymbol{g}_t - \frac{L\eta^2}{2}\|\boldsymbol{g}_t\|^2\right]. \quad (7)$$

Each stochastic gradient is an unbiased estimator of the full gradient, then $\mathbb{E}[\boldsymbol{g}_t] = \nabla F(\boldsymbol{w}_t)$. Moreover, for any random variable X, it holds $\mathbb{E}[X^2] = \mathbb{E}[X]^2 + \mathrm{Var}(X)$. Applying this relation to each component of the vector $\boldsymbol{g}_t$, and then summing up, we obtain:

$$\mathbb{E}[\|\boldsymbol{g}_t\|^2] = \|\nabla F(\boldsymbol{w}_t)\|^2 + \mathbb{V}(\boldsymbol{g}_{i,t})/k, \quad (8)$$

where $\mathbb{V}(\boldsymbol{g}_{i,t})$ denotes the sum of the variances of the different components of $\boldsymbol{g}_{i,t}$, i.e., $\mathbb{V}(\boldsymbol{g}_{i,t}) \triangleq \sum_{l=1}^{d} \mathrm{Var}([\boldsymbol{g}_{i,t}]_l)$. Notice that $\mathbb{V}(\boldsymbol{g}_{i,t})$ does not depend on i, because each worker has access to the complete dataset (assumption made in Sect. II, the same as in [11]). Then, combining (7) and (8), $\mathcal{G}_{k,t}$ can be rewritten as

$$\mathcal{G}_{k,t} = \left(\eta - \frac{L\eta^2}{2}\right)\|\nabla F(\boldsymbol{w}_t)\|^2 - \frac{L\eta^2}{2}\frac{\mathbb{V}(\boldsymbol{g}_{i,t})}{k}. \quad (9)$$

When full batch gradient descent is used, the optimal choice of the learning rate is $\eta = 1/L$, because it maximizes the expected gain. With this choice of the learning rate, Eq. (9) becomes:

$$\mathcal{G}_{k,t} = \frac{\eta}{2}\left(\|\nabla F(\boldsymbol{w}_t)\|^2 - \frac{\mathbb{V}(\boldsymbol{g}_{i,t})}{k}\right). \quad (10)$$

When the loss is not L-smooth, or the constant L is unknown, the learning rate is selected through some preliminary experiments (details in Sect. IV). We assume that (10) still holds.

Equation (10) shows that the gain increases as k increases. This corresponds to the fact that the more gradients are aggregated at the PS, the closer $-\boldsymbol{g}_t$ is to its expected value $-\nabla F(\boldsymbol{w}_t)$, i.e., to the steepest descent direction for the loss function. We also remark that the gain sensitivity to k depends on the relative ratio of $\mathbb{V}(\boldsymbol{g}_{i,t})$ and $\|\nabla F(\boldsymbol{w}_t)\|^2$, that keeps changing during the training (see for example Fig. 1). Correspondingly, we can expect that the optimal value of k will vary during the training process, even when computation and communication times do not change in the cluster. Experiments in Sect. IV confirm this is the case.

Computing the exact value of $\mathcal{G}_{k,t}$ would require the workers to process the whole dataset, leading to much longer iterations. We want rather to evaluate $\mathcal{G}_{k,t}$ with limited overhead for the workers. In what follows, we discuss how to estimate $\|\nabla F(\boldsymbol{w}_t)\|^2$ and $\mathbb{V}(\boldsymbol{g}_{i,t})$ to approximate $\mathcal{G}_{k,t}$ in (10). We first provide estimators that use information available *at the end* of iteration t, i.e., after k_t has been selected and the k_t fastest gradients have been received. Then, we build from these estimators new ones, that can be computed *at the beginning* of the iteration t and then can be used to select k_t. Given a quantity θ_t to be estimated at iteration t, we denote the first estimator as $\widehat{\theta_t}^+$ and the second one as $\widehat{\theta_t}$.

We start by estimating $\mathbb{V}(\boldsymbol{g}_{i,t})$ through the usual unbiased estimator for the variance:

$$\widehat{\mathbb{V}(\boldsymbol{g}_{i,t})}^+ = \sum_{l=1}^{d}\frac{1}{k_t - 1}\sum_{j=1}^{k_t}\left([\boldsymbol{g}_{j,t} - \boldsymbol{g}_t]_l\right)^2. \quad (11)$$

Next, we study the estimator of $\|\nabla F(\boldsymbol{w}_t)\|^2$. First, we can trivially use $\|\boldsymbol{g}_t\|^2$ to estimate $\mathbb{E}[\|\boldsymbol{g}_t\|^2]$, i.e., $\widehat{\mathbb{E}[\|\boldsymbol{g}_t\|^2]}^+ = \|\boldsymbol{g}_t\|^2$. Since $\|\nabla F(\boldsymbol{w}_t)\|^2 = \mathbb{E}[\|\boldsymbol{g}_t\|^2] - \mathbb{V}(\boldsymbol{g}_{i,t})/k_t$ (from (8)), we can estimate $\|\nabla F(\boldsymbol{w}_t)\|^2$ as follows

$$\widehat{\|\nabla F(\boldsymbol{w}_t)\|^2}^+ = \max\left(\widehat{\mathbb{E}[\|\boldsymbol{g}_t\|^2]}^+ - \frac{\widehat{\mathbb{V}(\boldsymbol{g}_{i,t})}^+}{k_t}, 0\right), \quad (12)$$

to guarantee non-negativity of the estimate.

Estimates in (11) and (12), cannot be computed at the beginning of iteration t, but it is possible to compute them for earlier iterations, and use these past estimates to predict the future value. DBW simply averages the past D estimates (or the first $t-1$ if $t \leq D$), i.e.,

$$\widehat{\mathbb{V}(\boldsymbol{g}_{i,t})} = \frac{1}{D}\sum_{v=1}^{D}\widehat{\mathbb{V}(\boldsymbol{g}_{i,t-v})}^+, \quad (13)$$

$$\widehat{\|\nabla F(\boldsymbol{w}_t)\|^2} = \frac{1}{D}\sum_{v=1}^{D}\widehat{\|\nabla F(\boldsymbol{w}_{t-v})\|^2}^+. \quad (14)$$

Combining (10), (13) and (14), the gain estimate is

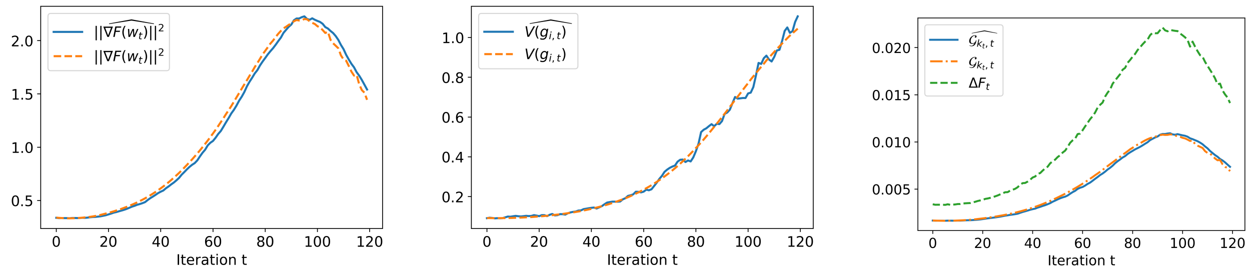
$$\widehat{\mathcal{G}_{k,t}} = \frac{\eta}{2}\left(\widehat{\|\nabla F(\boldsymbol{w}_t)\|^2} - \frac{\widehat{\mathbb{V}(\boldsymbol{g}_{i,t})}}{k}\right). \quad (15)$$

In Fig. 1, we show our estimates during one training process on the MNIST dataset (details in Sect. IV), where our algorithm (described below in Sect. III-C) is applied to dynamically choose k. The solid lines are the estimates given by (13), (14), and (15). The dashed lines present the exact values (we have instrumented our code to compute them). We can see from Figures 1(a) and 1(b) that the proposed estimates $\widehat{\|\nabla F(\boldsymbol{w}_t)\|^2}$ and $\widehat{\mathbb{V}(\boldsymbol{g}_{i,t})}$ are very accurate. Figure 1(c) compares the loss decrease ΔF_t (observed a posteriori) and $\widehat{\mathcal{G}_{k_t,t}}$. As expected $\widehat{\mathcal{G}_{k_t,t}}$ is a lower bound for ΔF_t, but the two quantities are almost proportional. This is promising, because if the lower bound $\widehat{\mathcal{G}_{k,t}}/T_{k,t}$ and the function $\Delta F_t/T_{k,t}$ were exactly proportional, their maximizers would coincide. Then, working on the lower bound, as we do, would not be an approximation.

B. Iteration Duration

In this subsection, we discuss how to estimate the time $T_{k,t}$ the PS needs to receive k gradients of $\boldsymbol{w}_t$ after the update $\boldsymbol{w}_t$ at iteration t. As in [19], we call *round trip time* the total (random) time an idle worker needs to 1) retrieve the new parameter vector, 2) compute the corresponding gradient, and 3) send it back to the PS.

When the PS starts a new iteration t ($t > 0$), there are k_{t-1} workers ready to compute the new gradient while the other $n - k_{t-1}$ workers are still computing stale gradients, i.e., relative to past parameter vectors



(a) Gradient norm

(b) Gradient variance

(c) Loss decrease

Fig. 1: Estimation of the loss decrease. MNIST, $n = 16$ workers, batch size $B = 500$, learning rate $\eta = 0.01$, estimates computed over the last $D = 5$ iterations.

$\boldsymbol{w}_{t-\tau}$ with $\tau > 0$. $T_{k,t}$ depends not only on the value of k but also on the value of k_{t-1} and the $n - k_{t-1}$ residual round trip times (i.e. the remaining times for the $n - k_{t-1}$ busy workers to complete their tasks). We assume that most of such dependence is captured by the number k_{t-1}. This would be correct if round trip times were exponential random variables. Let $\mathsf{t}_{h,i,t}$ denote the time the PS spends for receiving the i-th gradient of $\boldsymbol{w}_t$, provided that it has waited $k_{t-1} = h$ gradients at iteration $t - 1$. Under our assumptions, for given values of h and i, the values $\{\mathsf{t}_{h,i,t}\}$ can be seen as samples of the same random variable that we denote by $\mathcal{T}_{h,i}$. For estimating $T_{k,t}$, we consider $\widehat{T_{k,t}} = \widehat{\mathbb{E}[\mathcal{T}_{k,k}]}$.

Consider $k_{t-1} = h$ and $k_t = k$. The PS can collect the samples $\mathsf{t}_{h,i,t}$ for $i \leq k$ (it needs to wait k gradients before moving to the next iteration), but also for $i > k$ because late workers still complete the ongoing calculations. In fact, late workers may terminate the computation and send their (by now stale) gradients to the PS, before they receive the new parameter vector. Even if a new parameter vector is available at the local queue (and then they know their gradient is not needed), in DBW workers still notify the completion to the PS, providing useful information to estimate $T_{k,t}$ with limited communication overhead.

A first naive approach to estimate $\mathbb{E}[\mathcal{T}_{k,k}]$ is to average the samples obtained over the past history. But, actually, there is much more information that can be exploited to improve estimates, if we jointly estimate the complete set of values $\mathbb{E}[\mathcal{T}_{h,k}]$, for $h, k = 1, \ldots, n$. In fact, the following pathwise relation holds for each h and i: $\mathsf{t}_{h,i,t} \leq \mathsf{t}_{h,i+1,t}$, because i denotes the order of gradients' arrivals. As a consequence, $\mathbb{E}[\mathcal{T}_{h,i}] \leq \mathbb{E}[\mathcal{T}_{h,i+1}]$. Moreover, coupling arguments lead to conclude that $\mathbb{E}[\mathcal{T}_{h+1,i}] \leq \mathbb{E}[\mathcal{T}_{h,i}]$ and $\mathbb{E}[\mathcal{T}_{i,i}] \leq \mathbb{E}[\mathcal{T}_{i+1,i+1}]$. These two inequalities express the following intuitive facts: 1) if an iteration starts with more workers available to compute, the PS will collect i gradients faster (on average), 2) constantly waiting a smaller number of gradients leads to faster iterations. These inequalities allow us to couple the estimations of $\mathbb{E}[\mathcal{T}_{h,k}]$, for $h, k = 1, \ldots, n$. Samples for a given pair (h, k) can thus contribute not only to the estimation of $\mathbb{E}[\mathcal{T}_{h,k}]$ but also of other pairs. This is useful because the number of samples for different (h, k) is proportional to the number of times k_t has been selected equal to h. There can be many samples for a given pair and much less (even none) for another one.

Let $\mathbb{T}_{h,k,t}$ be the set of samples available up to iteration t for (h, k), i.e., $\mathbb{T}_{h,k,t} = \{\mathsf{t}_{h,k,t'}, \forall t' \leq t\}$. We propose to estimate $\{\mathbb{E}[\mathcal{T}_{h,k}], h, k = 1, \ldots, n\}$ by solving the following optimization problem:

$$\begin{aligned} \underset{x_{h,k}}{\text{minimize}} \quad & \sum_{h,k=1}^{n} \sum_{y \in \mathbb{T}_{h,k,t}} (y - x_{h,k})^2 && (16) \\ \text{subject to} \quad & x_{h,k} \leq x_{h,k+1}, && \text{for } k = 1, \ldots, n-1 \\ & x_{h+1,k} \leq x_{h,k}, && \text{for } h = 1, \ldots, n-1 \\ & x_{k,k} \leq x_{k+1,k+1}, && \text{for } k = 1, \ldots, n-1 \end{aligned}$$

Let $x^*_{h,k}$ be the solution of problem (16). Then, $\widehat{\mathbb{E}[\mathcal{T}_{h,k}]} = x^*_{h,k}$, $\forall h, k = 1, \ldots, n$ and we have $\widehat{T_{k,t}} = x^*_{k,k}$. We observe that, without the constraints, the optimal value $x^*_{h,k}$ at iteration t is the empirical average of the corresponding set $\mathbb{T}_{h,k,t}$. Hence, Problem (16) is a natural way to extend the empirical average estimators, while accounting for the constraints. For our application, the convex quadratic optimization problem (16) can be solved in polynomial time through solvers like CVX [20].

In Fig. 2, we compare our estimator with the naive empirical average. We observe that the naive method 1) cannot provide estimates for a given value h before it selects $k_t = h$, 2) leads often to estimates that are in the wrong relative order. By enforcing the inequality constraints, our estimator (16) is able to obtain more precise estimates, in particular for the values $k = 3$ and $k = 4$ that are tested less frequently in this experiment.

C. Dynamic Choice of k_t

DBW rationale is to select the parameter k_t that maximizes the expected decrease of the loss function per time unit, i.e.:

$$k_t = \underset{1 \leq k \leq n}{\arg\max} \frac{\widehat{\mathcal{G}_{k,t}}}{\widehat{T_{k,t}}}. \tag{17}$$

In most of the existing implementations of distributed gradient methods for ML (including PyTorch's one), each worker i can send to the PS the local average loss computed on its mini-batch. The PS can thus estimate the current loss as

$$\widehat{F}_t = \frac{1}{k_t} \sum_{i=1}^{k_t} \frac{1}{B} \sum_{x \in \mathbb{B}_i} f(x, \boldsymbol{w}_t).$$

The PS usually exploits this information to evaluate a stopping condition. DBW takes advantage of this available information to avoid decreasing k_t from one iteration to the other, when the loss appears to be increasing (and then we need more accurate gradient estimates, rather than noisier ones). We modify (17) to

$$k_t = \max\left(\underset{1 \leq k \leq n}{\arg\max} \frac{\widehat{\mathcal{G}_{k,t}}}{\widehat{T_{k,t}}},\ (k_{t-1} + 1) \cdot \mathbb{1}_{\substack{\hat{F}_{t-1} > \beta \hat{F}_{t-2} \\ \wedge\, k_{t-1} < n}} \right), \tag{18}$$

where $\beta \geq 1$ (we select $\beta = 1.01$ in our experiments) and $\mathbb{1}_A$ denotes the indicator function (equal to 1 iff A is true). If the loss has become β times larger since the previous iteration, then (18) forces $k_t \geq k_{t-1} + 1$.

IV. Experiments

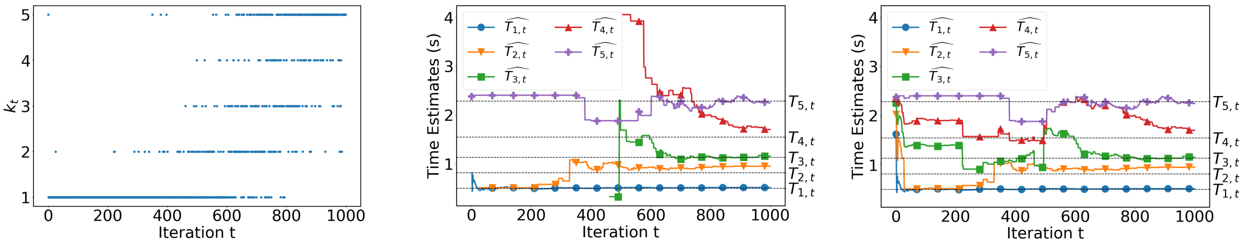
We have implemented DBW in PyTorch [13] using the MPI backend. The code is available [12]. The experiments have been run on a CPU/GPU cluster, with different GPUs available (e.g., Nvidia Tesla V100, GeForce GTX 1080 Ti and Titan X). In order to have a fine control over the round trip times, our code allows to generate computation and communication times according to different distributions (uniform, exponential, Pareto, etc.) or read from a trace provided as input file. The system operates at the maximum speed guaranteed by the underlying cluster, but it maintains a virtual clock to keep track of when events would have happened.

In what follows, we show that the optimal setting for the number of backup workers varies, not only with the round trip time distributions, but also with the hyper-parameters of the optimization algorithm like the batch size B. Moreover, the optimal setting depends as well on the stage of the training process, and then changes over time, even when the cluster is stationary (round trip times do not change during the training period).

In all experiments DBW achieves nearly optimal performance in terms of convergence time, and sometimes it even outperforms the optimal static setting, that is found through an exhaustive offline search over all values $k \in \{1, ..., n\}$. We also compare DBW with a variant where the gain $\mathcal{G}_{k,t}$ is not estimated as in (15), but it equals the number of aggregated gradients k, as proposed in [14]. We call this variant blind DBW (B-DBW), because it is oblivious to the current state of the training. We find that this approach is too simplistic: ignoring the current stage of the optimization problem leads to worse performance than DBW.

We evaluated DBW, B-DBW, and the static settings on different ML problems, including linear regression on synthetic and CT [21] datasets, and classification on MNIST [22], a dataset with 60000 images portraying handwritten digits. Results are qualitatively similar. Due to the page limit, we show only those for MNIST, for which we trained a neural network with two convolutional layers with 5×5 filters and two fully connected layers. The loss function was the cross-entropy one.

The learning rate is probably the most critical hyper-parameter in ML optimization problems. Ideally, it should be set to that largest value that still guarantees convergence. It is important to note that different static settings for the number of backup workers require different values for the learning rate. In fact, the smaller is k, the noisier is the aggregate gradient $\boldsymbol{g}_t$, so that the smaller should be the learning rate. The rule of thumb proposed in the seminal paper [11] is to set the learning rate proportional to k, i.e. $\eta(k) \propto k$. This corresponds to the standard recommendation to have the learning rate proportional to the (aggregate) batch size [23], [24]. In static settings, aggregating k gradients is equivalent to use a batch size equal to kB, so that the learning rate should scale accordingly. An alternative approach is to tune the learning rate independently for each static value of k according to the empirical rule in [25], that requires to run a number of experiments and determine the inflection points of a specific curve. This rule leads as well to learning rates increasing with k. We call the two settings respectively the *proportional* and the *knee* rule. The maximum learning rate for the proportional rule is set equal to the value determined for $k_t = n$ by the knee rule. The same value is also used as learning rate for DBW and B-DBW, independently from the specific



(a) Values of k selected.

(b) Empirical average.

(c) Constraint-aware estimator (16).

Fig. 2: Estimation of $T_{k,t}$. $n = 5$ workers.

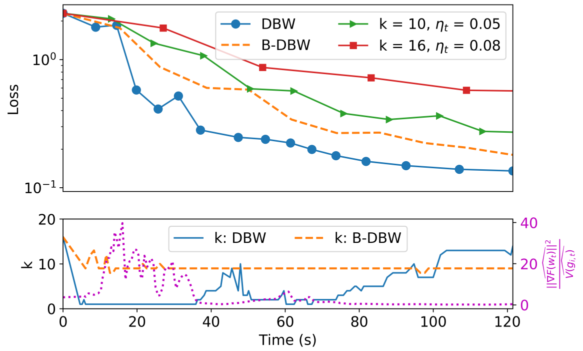


Fig. 3: Loss versus time. MNIST, batch size $B = 500$, $n = 16$ workers, estimates computed over the last $D = 5$ iterations, proportional rule with $\eta(k) = 0.005k$, round trip times follow shifted exponential distribution $0.3 + 0.7\text{Exp}(1)$.

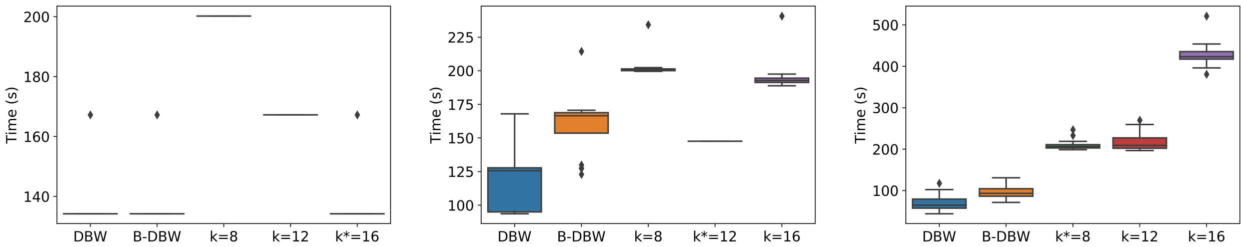
value they select for k_t. In fact, DBW and B-DBW can safely operate with a large learning rate because they dynamically increase k_t up to n, when they detect that the loss is increasing.

Figure 3 shows, for a single run of the training process, the evolution of the loss over time and the corresponding choices of k_t for the two dynamic algorithms. For static settings, the learning rate follows the proportional rule and the optimal static setting is $k^* = 10$. We can see that DBW achieves the fastest convergence across all other tested configurations of k, by using a different value of k in different stages of the training process. In fact, as we have discussed after introducing (10), the effect of k on the gain depends on the module of the gradient and on the variability of the local gradients. In the bottom subplot, the dotted line shows how their ratio varies during the training process. Up to iteration 40, $\mathbb{V}(\boldsymbol{g}_{i,t})$ is negligible in comparison to $\|\nabla F(\boldsymbol{w}_t)\|^2$. DBW then selects small values for k_t loosing a bit in terms of the gain, but significantly speeding up the duration of each iteration by only waiting for the fastest workers. As the parameter vector approaches a local minimum, $\|\nabla F(\boldsymbol{w}_t)\|^2$ approaches zero, and the gain becomes more and more sensitive to k, so that DBW progressively increases k_t up to reach $k_t = n = 16$ as shown by the solid line. On the contrary B-DBW (the dashed line) selects most of the time $k_t = 9$ with some variability due to the randomness of the estimates $\widehat{T_{k,t}}$.

A. Round trip time effect

In this subsection we consider round trip times (see Sect. III-B) are i.i.d. according to a shifted exponential random variable $1 - \alpha + \alpha \times \text{Exp}(1)$, where $0 \leq \alpha \leq 1$. We consider later realistic time distributions. This choice, common to [19], [26], allows us to easily tune the variability of the round trip times by changing α. When $\alpha = 0$, all gradients arrive at the same time at the PS, so that the PS should always aggregate all of them. As α changes from 0 to 1, the variance of the round trip times increases, and waiting for $k < n$ gradients becomes advantageous.

Figure 4 compares the time needed to reach a training loss smaller than 0.2 for the two dynamic algorithms and the static settings $k = 16$, $k = 12$, and $k = 8$, that are optimal respectively for $\alpha = 0$, $\alpha = 0.2$, $\alpha = 1$. For each of them, we carried out 20 independent runs with different seeds. We find that our dynamic algorithm achieves the fastest convergence in all the three scenarios, it is even 1.2x faster and 3x faster than the optimal static settings for $\alpha = 0.2$ and $\alpha = 1$. There are two factors that determine this observation. First, as discussed for Fig. 3, there is no unique optimal value of k to be used across the whole training process, and DBW manages to select the most indicated value in different stages of the training process. Second, DBW takes advantage of a larger learning rate. Both factors play a role. For example if we focus on Fig. 4(c), the learning rate for DBW is twice faster than

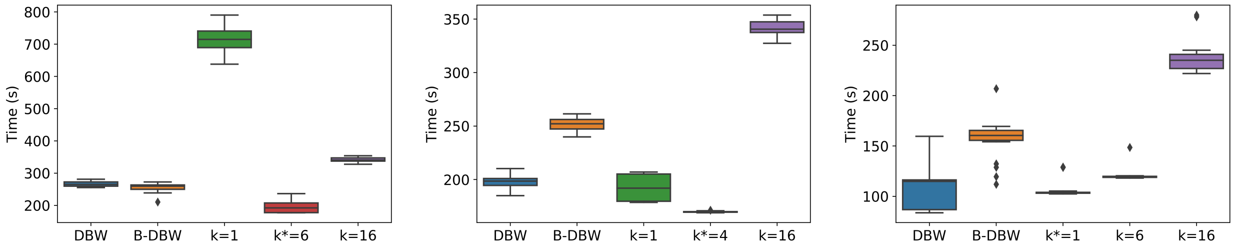


(a) $\alpha = 0$

(b) $\alpha = 0.2$

(c) $\alpha = 1$

Fig. 4: Effect of round trip time distribution. MNIST, $n = 16$ workers, batch size $B = 500$, estimates computed over the last $D = 5$ iterations, proportional rule for $\eta(k)$ in static settings where $\eta(k) = 0.005k$.



(a) $B = 16$, $\eta \in \{0.01, 0.045, 0.05\}$

(b) $B = 128$, $\eta \in \{0.04, 0.044, 0.05\}$

(c) $B = 500$, $\eta \in \{0.07, 0.07, 0.08\}$

Fig. 5: Effect of batch size B. MNIST, $n = 16$ workers, estimates computed over the last $D = 5$ iterations, knee rule for η in static settings with values shown above for each k.

that for $k = 8$, but DBW is on average 3x faster. Then, adapting k achieves an additional 1.5x improvement. The importance of capturing the dynamics of the optimization process is again also evident by comparing DBW with B-DBW. While B-DBW takes advantage of a higher learning rate as well, it does not perform as well as our solution DBW.

B. *Batch size effect*

The batch size B is another important hyper-parameter. It is often limited by the memory available at each worker, but can also be determined by generalization performance of the final model [27]. In this subsection we highlight how B also affects the optimal setting for k. These findings confirm that configuring the number of backup workers is indeed a difficult task, and knowing the characteristics of the underlying cluster is not sufficient.

The experiments differ in two additional aspects from those in Fig. 4. First, the distribution of the round trip times is taken from a real ML experiment on a Spark cluster (the distribution is similar to [19, Fig. 7]). Second, learning rates are configured according to the knee rule. We observe that the knee rule leads to a weaker variability of the learning rate in comparison to the proportional rule: for example, for $B = 16$, η increases by less than a factor 5 when k changes from $k = 1$ to $k = 16$, and it increases much less for larger B.

Figure 5 shows the results for $B = 16, 128, 500$, comparing the dynamic methods with a few static settings, including the optimal static one that decreases from $k^* = 6$ for $B = 16$ to $k^* = 1$ for $B = 500$. Again, Equation (10) helps to understand this change of the optimal static setting with different batch size: as the batch size increases, the variability of gradients decreases, so that the numerator depends less on k. The advantage of reducing $T_{k,t}$ by selecting a small k can compensate the corresponding decrease of the gain $\mathcal{G}_{k,t}$.

Since learning rates chosen by the knee rule for the static settings are now close to dynamic ones, DBW does not outperform the optimal static setting, but its performance are quite close, and significantly better than B-DBW for $B = 128, 500$. It is worthy to stress that, when running a given ML problem on a specific cluster environment, the user cannot predict the optimal static setting k^* without running preliminary short training experiments for every k. DBW does not need them.

C. *Robustness to slowdowns*

Until now, we have considered a stationary setting where the distribution of round trip times does not change

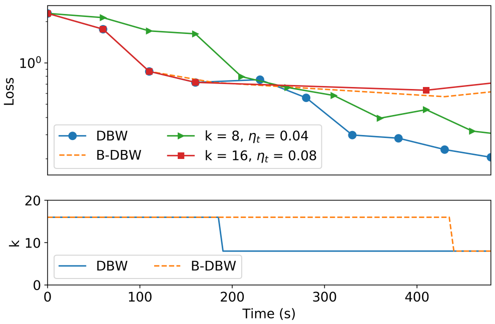


Fig. 6: Robustness to slowdowns of the system. MNIST, $n = 16$ workers, batch size $B = 500$, estimates computed over the last $D = 5$ iterations, proportional rule for $\eta(k)$ in static settings where $\eta(k) = 0.005k$.

during the training. Figure 6 shows an experiment in which half of the workers experience a sudden slowdown during the training process. Initially, round trip times are all equal and deterministic, so that the optimal setting is $k_t = n = 16$. Suddenly, at time $t = 160$s, half of the workers in the clusters slow down by a factor 5 and the optimal static configuration is now to select $k_t = n/2 = 8$. We can see that DBW detects the slowdowns in the system and then correctly selects $k_t = 8$.

V. Conclusions

In this paper, we have shown that the number of backup workers needs to be adapted at run-time and the correct choice is inextricably bounded, not only to the cluster's configuration and workload, but also to the hyper-parameters of the learning algorithm and the stage of the training. We have proposed a simple algorithm DBW that, without priori knowledge about the cluster or the problem, achieves good performance across a variety of scenarios, and even outperforms in some cases the optimal static setting. As a future research direction, we want to extend the scope of DBW to dynamic resource allocation, e.g. by automatically releasing computing resources if $k_t < n$ and the fastest k_t gradients are always coming from the same set of workers.

In general, we believe that distributed systems for ML are in need of adaptive algorithms in the same spirit of the utility-based congestion control schemes developed in our community starting from the seminal paper [28]. As our paper points out, it is important to define new utility functions that take into account the learning process. Adaptive algorithms are even more needed in the federated learning scenario [29], where ML training is no more relegated to the cloud, but it occurs in the wild over the whole internet. Our paper shows that even simple algorithms can provide significant performance improvements.

References

[1] K. Canini *et al.*, "Sibyl: A system for large scale supervised machine learning," 2014, technical talk.

[2] A. Harlap *et al.*, "Addressing the straggler problem for iterative convergent parallel ml," in *7th ACM SoCC*, 2016, pp. 98–111.

[3] G. Neglia *et al.*, "The role of network topology for distributed machine learning," in *INFOCOM*, 2019, pp. 2350–2358.

[4] Y. Bao *et al.*, "Deep learning-based job placement in distributed machine learning clusters," in *INFOCOM*, 2019, pp. 505–513.

[5] C. Chen *et al.*, "Round-robin synchronization: Mitigating communication bottlenecks in parameter servers," in *INFOCOM*, 2019, pp. 532–540.

[6] M. Li *et al.*, "Scaling distributed machine learning with the parameter server," in *11th USENIX OSDI*, 2014, pp. 583–598.

[7] G. Ananthanarayanan *et al.*, "Effective straggler mitigation: Attack of the clones," in *10th USENIX Conf. NSDI*, 2013, pp. 185–198.

[8] C. Karakus *et al.*, "Straggler mitigation in distributed optimization through data encoding," in *Proc. of NIPS*, 2017, pp. 5434–5442.

[9] S. Li *et al.*, "Near-optimal straggler mitigation for distributed gradient methods," in *IEEE IPDPS*, 2018, pp. 857–866.

[10] W. Dai *et al.*, "Toward understanding the impact of staleness in distributed machine learning," in *7th ICLR*, 2019.

[11] J. Chen *et al.*, "Revisiting distributed synchronous sgd," in *ICLR Workshop Track*, 2016.

[12] "DBW," https://gitlab.inria.fr/chxu/dbw.

[13] "PyTorch," https://pytorch.org/.

[14] M. Teng *et al.*, "Bayesian distributed stochastic gradient descent," in *Advances in NIPS 31*, 2018, pp. 6378–6388.

[15] L. Balles *et al.*, "Coupling adaptive batch sizes with learning rates," in *Proc. of the 33th Conference on UAI*, 2017.

[16] H. Wang *et al.*, "Distributed machine learning with a serverless architecture," in *INFOCOM*, 2019, pp. 1288–1296.

[17] S. Bubeck, "Convex optimization: Algorithms and complexity," *Found. Trends Mach. Learn.*, vol. 8, no. 3-4, pp. 231–357, 2015.

[18] L. Bottou *et al.*, "Optimization methods for large-scale machine learning," *Siam Review*, vol. 60, no. 2, pp. 223–311, 2018.

[19] K. Lee *et al.*, "Speeding up distributed machine learning using codes," *IEEE Transactions on Information Theory*, vol. 64, no. 3, pp. 1514–1529, March 2018.

[20] M. Grant *et al.*, "CVX: Matlab software for disciplined convex programming, version 2.1," http://cvxr.com/cvx, 2014.

[21] F. Graf *et al.*, "2d image registration in ct images using radial image descriptors," in *MICCAI*, 2011, pp. 607–614.

[22] "MNIST database," http://yann.lecun.com/exdb/mnist/.

[23] P. Goyal *et al.*, "Accurate, large minibatch SGD: training imagenet in 1 hour," *CoRR*, vol. abs/1706.02677, 2017.

[24] S. L. Smith *et al.*, "Don't decay the learning rate, increase the batch size," in *ICLR*, 2018.

[25] L. N. Smith, "Cyclical learning rates for training neural networks," in *Winter Conference on WACV*, 2017, pp. 464–472.

[26] S. Dutta *et al.*, "Slow and stale gradients can win the race: Error-runtime trade-offs in distributed SGD," in *AISTATS*, 2018, pp. 803–812.

[27] E. Hoffer *et al.*, "Train longer, generalize better: closing the generalization gap in large batch training of neural networks," in *Advances in NIPS 30*, 2017, pp. 1731–1741.

[28] F. P. Kelly *et al.*, "Rate control for communication networks: shadow prices, proportional fairness and stability," *Journal of the Operational Research society*, vol. 49, no. 3, pp. 237–252, 1998.

[29] J. Konecný *et al.*, "Federated optimization: Distributed optimization beyond the datacenter," in *NIPS (workshop)*, 2015.