Accelerating Distributed Learning in Non-Dedicated Environments

Chen Chen, *Member, IEEE*, Qizhen Weng, *Student Member, IEEE*, Wei Wang, *Member, IEEE*, Baochun Li, *Fellow, IEEE*, and Bo Li, *Fellow, IEEE*

Abstract—Machine learning (ML) models are increasingly trained with distributed workers possessing heterogeneous resources. In such scenarios, model training efficiency may be negatively affected by *stragglers*—workers that run much slower than others. Efficient model training requires eliminating such stragglers, yet for modern ML workloads, existing load balancing strategies are inefficient and even infeasible. In this article, we propose a novel strategy, called *semi-dynamic load balancing*, to eliminate stragglers of distributed ML workloads. The key insight is that ML workers shall be load-balanced at *iteration boundaries*, being non-intrusive to intra-iteration execution. Based on it we further develop LB-BSP, an integrated worker coordination mechanism that adapts workers' load to their instantaneous processing capabilities—by right-sizing the sample batches at the synchronization barriers. We have designed distinct load tuning algorithms for ML in CPU clusters, in GPU clusters as well as in federated learning setups, based on their respective characteristics. LB-BSP has been implemented as a Python module for ML frameworks like TensorFlow and PyTorch. Our EC2 deployment confirms that LB-BSP is practical, effective and light-weight, and is able to accelerating distributed training by up to 54 percent.

Index Terms—Distributed machine learning, synchronization, load balancing, federated learning

1 Introduction

Machine learning (ML) models, such as deep neural networks, are widely used for a range of applications to attain state-of-the-art performance [1], [2], [3], [4]. Owing to their compute-intensive nature, ML models are often trained in a *distributed* manner [5], [6], [7], [8]: many *worker* threads iteratively process small subsets of the training data (i.e., *sample batches*), and use the computed updates to refine the global model parameters.

With the surge of ML training demands, it has become increasingly common to serve ML jobs in *non-dedicated* environments, e.g., with *heterogeneous* hardware from spot markets [9], or with *time-varying* resources from shared production clusters [10], [11], [12]. In such cases, workers with less capable resources would progress slower and become *stragglers*. Under the popular *Bulk Synchronous Parallel* (BSP) scheme, fast workers have to wait for slow ones at the end of each iteration, and stragglers would thus impair the model training efficiency.

To mitigate the negative effect of stragglers, a large number of mechanisms have been developed in the literature. For example, some [5], [13], [14] have proposed to relax the synchronization barriers to avoid end-of-iteration resource wastage, yet this negatively affects the update quality and requires more iterations for models to converge. In fact, stragglers in non-dedicated clusters are mainly caused by the *mismatch* between workers' load and their processing capability. In response, the most effective strategy to eliminate stragglers is to balance the load of workers according to their instantaneous processing capability.

Nonetheless, ML workloads pose new challenges to the load balancing community. Existing load balancing schemes designed for traditional multi-core or big data scenarios can be broadly classified into two categories: *static* and *dynamic* load balancing (Section 2.2.3). Yet, different from those traditional workloads such as HPC processing [15], [16] or MapReduce [17], ML computations are highly *structured* (with thousands of short iterations) and also *tensor-based* (samples in each iteration are packaged into a *non-divisible* matrix for fast processing). Static load balancing approaches [18], [19], [20] are not aware of runtime resource variations, and dynamic approaches [21], [22], [23], which are mainly based on *work stealing* [15], [16], [24], are also deficient for ML workloads. First, work stealing usually requires fine-grained worker progress monitoring and runtime load migration, which is inefficient for an iterative model training process. Second, runtime load migration assumes that the samples be processed *one by one*, yet such a sequential manner hinders the full utilization of hardware accelerators like GPUs that are common for model ML frameworks [8], [25], [26].

In this paper, we design a new load balancing strategy for distributed model training workloads, called *semi-dynamic load balancing*. In broad strokes, the high-level idea is to perform all load balancing actions—worker

• *Chen Chen, Qizhen Weng, Wei Wang, and Bo Li are with the Department of Computer Science and Engineering, Hong Kong Science and Technology, Hong Kong. E-mail: {cchenam, qwengaa}@connect.ust.hk, {weiwa, bli}@cse.ust.hk.*
• *Baochun Li is with the Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON M5S, Canada. E-mail: bli@ece.toronto.edu.*

Manuscript received 7 Feb. 2021; revised 16 May 2021; accepted 20 July 2021. Date of publication 5 Aug. 2021; date of current version 8 Mar. 2023.
(Corresponding author: Chen Chen.)
Recommended for acceptance by S. Guo.
Digital Object Identifier no. 10.1109/TCC.2021.3102593

monitoring, straggler identification and load redistribution—on the iteration boundaries, with load adjustment enforced by tuning each worker's sample *batch size*. Following such a high-level idea, we propose *Load-Balanced Bulk Synchronous Parallel* (LB-BSP), a composite scheme atop BSP that seeks to equalize all the workers' batch processing times by speculatively configuring their batch sizes at the synchronization barriers. An immediate question is then how to set the batch size at the synchronization barriers for the best load balancing effect in the upcoming iteration. This evolves into different challenges for CPU and GPU clusters that are two typical platform types for distributed learning.

For a CPU worker, our profiling work (Section 3.2.1) shows that its batch processing time is proportional to its batch size, with the proportion coefficient representing the instantaneous sample processing capability. In shared clusters, such a coefficient may vary drastically with the temporal resources [12], [27], and thus shall be predicted prior to each iteration. For this purpose, we employ a special kind of recurrent neural network called NARX [28], which can make accurate performance predictions by taking into account the driving resources such as CPU and memory.

In multi-tenant GPU clusters [10], [11], [29], [30], a GPU is usually dedicated to one worker without sharing. But the relationship between a GPU worker's batch processing time and batch size is not proportional and difficult to profile at runtime (Section 3.3.1). Therefore, instead of the profiling-based analytical methods, we propose an *iterative* batch size tuning algorithm that can efficiently approximate the load-balanced state without prior knowledge.

Furthermore, while our method of worker-adaptive batch sizing can effectively eliminate stragglers, it inevitably results in *non-uniform* batch sizes among different workers, which may negatively affect the training accuracy. To ensure that model training process can still converge correctly even with inconsistent batch sizes, we further propose *weighted gradient aggregation*, in which the batch size of each worker is used as the weight of its gradient in aggregation.

Moreover, since a preliminary version of this work [31] only focuses on cluster environments yet neglects the rapidly-emerging trend of practicing ML with edge devices, in this paper we extend our LB-BSP solution for *federated learning* (FL) setups. FL is a popular paradigm that allows edge clients to collaborate in model training without revealing their local private data [32], [33], [34]. It adopts a special training algorithm called FedAvg, which includes multiple local iterations in each synchronization round. Meanwhile, samples on different clients are usually of different distributions and cannot be migrated under the privacy requirement of FL. Our extended LB-BSP design complies with these properties, and can saliently speed up model convergence with negligible accuracy loss.

We have implemented LB-BSP with a Python module that can be easily integrated into existing ML frameworks like TensorFlow [8], PyTorch [26] and MXNet [25]. Our experiments on Amazon EC2 with popular benchmark training workloads show that LB-BSP can effectively eliminate stragglers in non-dedicated clusters, achieving near-optimal training efficiency with negligible overhead. In particular, in a 16-node heterogeneous GPU cluster, LB-BSP outperforms existing worker coordination schemes by over 54 percent; and in a 32-node shared CPU cluster, it attains an improvement of up to 38.7 percent over existing straggler mitigating approaches.

2 Background and Motivation

2.1 Research Background

Basics of Machine Learning. Given a machine learning (ML) model, the objective of model training process is to find the *model parameters* ω^* that can minimize the *loss function* $L(\omega)$ over the labeled training dataset S, i.e.,

$$\omega^* = \arg\min_{\omega} L(\omega) = \arg\min_{\omega} \frac{1}{|S|}\sum_{s\in S} l(s,\omega). \tag{1}$$

Here, $l(s,\omega)$ is the loss value for a data sample s in S.

A popular training algorithm is *mini-batch Stochastic Gradient Descent* [5], [35], or simply SGD.[1] Its basic idea is to iteratively refine model parameters ω with the gradients g calculated from the sample *batches*, i.e.,

$$\omega^{k+1} = \omega^k - \eta g^k, \text{ where } g^k = \frac{1}{|B^k|}\sum_{s\in B^k} \nabla l(s,\omega^k). \tag{2}$$

Here, k is the iteration number, B^k is a batch randomly sampled from the training dataset S, and η is the *learning rate*. Such training iterations would repeat until the model parameters ω finally converge.

Given the stochastic nature of SGD, the relationship between training completion (i.e., model convergence) and the sample processing amount is not deterministic, and the model training efficiency is usually decoupled into two parts: *hardware efficiency*—how fast each iteration can be finished, and *statistical efficiency*—how many iterations it takes towards model convergence. Therefore, efficient model training requires both high hardware efficiency and high statistical efficiency.

Distributed Model Training. To accelerate the model training process, it has been increasingly common to train ML models in a distributed manner, with a pool of parallel workers. Those workers typically collaborate in a synchronous mode (i.e., BSP) [36], [37], [38], under the *Parameter Server* (PS) [6], [13], [38] or *All-Reduce* [37] architecture.

With the widespread adoption of ML techniques, distributed model training is now conducted in various cluster environments, and we broadly classify them into two types: *dedicated* and *non-dedicated* clusters.

(1) Dedicated clusters implies that the clusters are homogeneous, and are composed of cutting-edge GPUs dedicated to one training job. For example, Facebook's well-known work of training ImageNet in one hour [37] is conducted in such a dedicated cluster with 256 Tesla P100 GPUs. While dedicated clusters can yield high training efficiency, they are expensive to maintain and not widely accessible [37], [39].

(2) Non-dedicated clusters refer to the clusters that are composed of heterogeneous hardware or shared by multiple

1. For simplicity, we focus on standard SGD in this paper. Yet our LB-BSP solution can also be applied to all SGD variants (e.g., Adam or RMSProp, which make use of model gradients in different manners), because LB-BSP itself does not compromise the aggregated gradients.

tenants. Since newer CPU or GPU generations get released at a rapid pace, a budget-limited user may choose to train models with a set of heterogeneous hardware (e.g., GPUs) from the local inventory [40], [41], or from the spot markets such as Amazon EC2 [9]. Meanwhile, large companies [10], [12] may host multiple ML jobs in their production clusters with mixed hardware types; for fairness, users may be allocated heterogeneous hardware [42], [43], [44], and their resource allocation may also be dynamically adjusted by cluster schedulers for resource packing purpose [11], [30]. Moreover, in recent years federated learning (FL) [32], [33], [34], [45] has emerged as a very promising solution that enables multiple clients to join their computing resources and local data in model training with data privacy preserved. A FL setup is essentially a non-dedicated cluster, because without centralized resource allocation, clients' processing capabilities are often inconsistent.

While model training in non-dedicated clusters is increasingly common, it is far less efficient than training in dedicated clusters due to the straggler problem, which is more severe in non-dedicated clusters. For clarity, we classify stragglers into two groups: *non-deterministic* and *deterministic* stragglers.

(1) Non-deterministic stragglers are caused by temporary disturbance like OS jitter or garbage collection, usually being transient and slight. They occur and vanish naturally in both dedicated and non-dedicated clusters.

(2) Deterministic stragglers are caused by heterogeneous worker resource quality or quantity, often severe and long-lasting. They occur only in non-dedicated clusters.

Compared with non-deterministic stragglers, deterministic stragglers are more harmful to distributed model training, by forcing fast workers to always wait for the slowest one in each iteration if under the popular BSP scheme. With the increasing prevalence of non-dedicated clusters, it is thus in urgent need to tame such deterministic stragglers.

2.2 Related Work

Stragglers have long been a thorny problem in distributed computing systems, and in the research literature there have been many attempts to tackle such a problem.

2.2.1 Bypassing Stragglers With Relaxed Synchronization

To exempt fast workers from waiting for stragglers, two worker coordinating schemes with the synchronization constraint compromised have been proposed: ASynchronous Parallel (ASP), and Stale Synchronous Parallel (SSP).

ASP. In ASP [5], workers can independently proceed to the next iteration without waiting for others. In this way, ASP wastes *no* compute cycles and attains high hardware efficiency. However, the price paid is low statistical efficiency: without global synchronization, the gradient computation often uses *stale* parameters, which yields low-quality updates and requires more iterations to converge [13], [46].

SSP. SSP [13], [14] comes as a middle ground between BSP and ASP. In SSP, fast workers wait for the stragglers *only when* the parameter *staleness* reaches a particular threshold. SSP can then accelerate ML iterations while providing a convergence guarantee. Nonetheless, SSP focuses primarily on non-deterministic stragglers, expecting the straggling workers to catch up soon in later iterations. This works for homogeneous clusters, but in non-dedicated clusters the deterministic stragglers may persist across many consecutive iterations, and the staleness quota of SSP can be quickly used up. After the quota is used up, fast workers will have to wait for the slowest ones in almost every iteration, leading to *low hardware efficiency*.

2.2.2 Mitigating Stragglers by Redundant Execution

Redundant execution [47], [48] is widely adopted to mitigate stragglers in traditional data analytics frameworks like MapReduce [17] and Spark [49]. It launches multiple copies of a straggling task and accepts results only from the one that finishes first. It has also been recently introduced to the context of distributed machine learning: Chen *et al.* [50] proposed to train deep neural models with extra backup workers and to use gradients from those workers finishing the earliest. However, redundant execution is merely a suboptimal solution: it mitigates some worst-case stragglers but fails to eliminate all of the stragglers *completely*. Even worse, backup workers themselves would consume additional resources.

2.2.3 Eliminating Stragglers by Load Balancing

The solutions we have discussed so far are agnostic to the root causes of stragglers, and they do not seek to prevent stragglers from occurring. As we mentioned, efficiency loss in non-dedicated clusters is primarily caused by deterministic stragglers, whose root cause is very clear—the mismatch between the workers' loads and their processing capabilities (existing ML frameworks [8], [25], [51] blindly assign a *uniform* batch load to all the workers). Therefore, to fundamentally eliminate such deterministic stragglers related to load-resource mismatching, we should resort to *load balancing* techniques. Load balancing is a classical research topic [52] in the parallel processing community, and existing solutions can be broadly classified into two types: *static* and *dynamic* load balancing.

Static Load Balancing. Static load balancing strategies [18], [19], [20], [53] set a constant load for each worker—as the execution commences—under a given scheme like Round-Robin [19] or with some static knowledge of the worker status [20], [53]. It does not require real-time progress measurements or cross-worker communication, but cannot react to resource variations that are common in non-dedicated clusters.

Dynamic Load Balancing. Dynamic load balancing strategies use *work-stealing* or *work-shedding* to redistribute load from heavily-loaded workers to lightly-loaded ones at runtime [15], [16], [21], [22], [24]. They are mostly developed for traditional *task*-based parallel programming models in multi-core or HPC systems. Recently, FlexRR [23] adopted such a dynamic strategy to tackle (non-deterministic) stragglers for ML workloads. It measures the workers' instantaneous progress at a fine granularity (100 checks per iteration); once a straggling worker lags behind others over a given threshold, it would yield certain sample processing load to a faster worker. Responding to dynamic changes of resources, dynamic load balancing schemes usually outperform static ones, but the cost paid is much higher computation and communication overhead (for progress monitoring, status collection and workload migration), which is not desirable in resource-intensive ML training. To reduce such overhead, FlexRR conducts straggler detection and load migration within designated *worker groups*, which nonetheless leads to suboptimal load balancing performance due to its lack of global coordination.

Moreover, a key assumption of dynamic load balancing strategies is that, workloads shall be processed in a *sequential* manner so that they can be arbitrarily split and transferred at runtime. However, this is not true for modern ML workloads. Training ML models is compute-intensive, and parallel-processing accelerators like GPUs are commonly used in practice, for which sequential processing is highly inefficient (due to the fixed accelerator launching overhead that is independent to batch size, as will be shown later in Fig. 4 of Section 3.3.1). To fully exploit the power of such accelerators, mainstream ML frameworks wrap all the samples in a batch as a tensor matrix (e.g., a `Tensor` in TensorFlow/PyTorch, or an `NDArray` in MXNet), which is concurrently processed *in a single round*. Given such *all-or-nothing* processing, it is hard to measure fine-grained worker progress or adjust its load in the midst of an iteration, making dynamic load balancing simply infeasible.

2.3 Semi-Dynamic Load Balancing

Objectives. Based on our discussions so far, our objective is to develop a load balancing strategy for ML workloads with the following properties:

(1) *Practicality.* It should be compatible with the style of tensor-based processing in existing ML frameworks.

(2) *Effectiveness.* It should be aware of the instantaneous worker execution status, and adjust the workers' load in a coordinated manner to attain the best load balancing effect.

(3) *Efficiency.* It should not interfere with regular processing within each iteration, and should try to be lightweight by avoiding cross-worker data movement.

Design Philosophy. To meet these objectives, we propose a new load balancing strategy called *semi-dynamic load balancing*. Tailored for ML workloads, its basic idea is to have workers' load be *static* within each iteration but *dynamic* across different iterations. In particular, we offload all load balancing operations—status measurements, straggler detection and load adjustment—at the *iteration boundaries of BSP*, using the *batch size* as a tool for load tuning. This strategy is feasible and can satisfy all of our design objectives, which we rationalize from the following three aspects:

(1) Measuring Worker Status at Iteration Boundaries. Training iterations in modern ML frameworks are relatively short (in *seconds* or even *sub-seconds*, as we show later in Figs. 1 and 4), and these iterations share a high similarity because an identical computation graph is used in each iteration. Therefore, the execution status in recent iterations is a valuable reference for that of the near future, relieving the need for costly intra-iteration progress measurements.

(2) Detecting stragglers at BSP iteration boundaries. Under BSP there is a synchronization barrier at the end of each iteration, offering a natural opportunity to centralize all the workers' status information and optimize, with a global view, their load for the best load balancing effect.

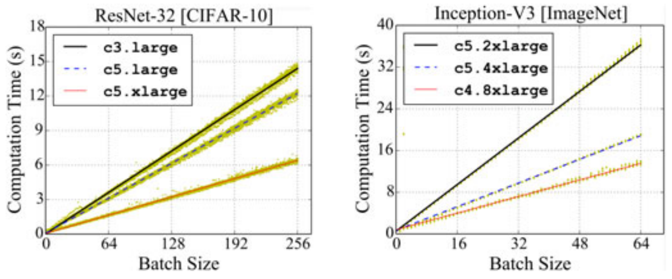


Fig. 1. The relationship between computation time and batch size for different EC2 CPU instances.

(3) Adjusting load by tuning the batch size at iteration boundaries. The batch size is a hyper-parameter that, as each iteration commences, dictates how many samples should be encapsulated into a tensor batch for processing in that iteration. It accurately controls a worker's load because each sample consumes identical compute cycles. Besides, given the stochastic (i.e., *sample-insensitive*) nature of SGD, load migration can be realized by increasing the batch size of one worker and reducing that on another. This can avoid the communication overhead without compromising the training convergence.

In the next section, we show how the philosophy of semi-dynamic load balancing can be implemented in practice.

3 LB-BSP

In this section, we present *Load-Balanced Bulk Synchronous Parallel* (LB-BSP), an integrated scheme atop BSP, for efficient distributed learning in non-dedicated clusters. We start with the problem formulation of LB-BSP, and then respectively elaborate our solutions for CPU clusters, GPU clusters and federated learning setups.

3.1 Problem Formulation

In each model training iteration, given the sample batch, a worker first calculates a local gradient and then remotely refines the global model. Here we refer to the entire duration as the *batch processing time*, denoted by t. It can be divided into two parts: *computation time* (t^p), which measures the time taken to compute the gradient, and *communication time*[2] (t^m), which measures the time taken for data transmission.

In a nutshell, LB-BSP seeks to equalize t on different workers by rightsizing their sample batches at the synchronization barriers. This can be formulated as an optimization problem. Given n workers with the initial batch size $\dot{x}$, we want to find the worker batch sizes $\vec{x} = (x_1, x_2, \ldots, x_n)$ that can minimize the longest batch processing time among all workers, i.e.,

2. Communication and computation are partially overlapped [36], [37], [38] in existing ML frameworks (e.g., TensorFlow, MXNet); for clarity, communication time in our definition excludes the overlapped periods.

TABLE 1
Summary of Important Notations

t	batch processing time	x_i	batch size on worker-i
t^p	computation time	$\dot{x}$	initial batch size
t^m	communication time	X	$n\dot{x}$ (n is worker number)
$\Gamma(\cdot)$	function between t^p and x	v	sample processing speed

$$\begin{aligned} \min_{\vec{x}=(x_1,x_2,\ldots,x_n)} \quad & \max_{i\in\{1,2,\ldots n\}} t_i, \\ \text{s.t.} \quad & t_i = t_i^p + t_i^m,\ i = 1,\ldots,n; \\ & t_i^p = \Gamma_i(x_i),\ i = 1,\ldots,n; \\ & \sum_{i=1}^{n} x_i = X = n\dot{x}. \end{aligned} \tag{3}$$

Here $\Gamma_i(\cdot)$ is the function between worker-i's computation time t_i^p and its batch size x_i. The last constraint ensures that the total number of samples processed in each iteration remains the same as that in BSP. Table 1 summarizes the important notations used in the paper.

Solution Overview. Solving problem (3) poses different challenges to CPU and GPU clusters. In CPU clusters, t_i increases linearly with x_i, but that ratio varies with the temporal resources. In GPU clusters, the relationship between t_i and x_i is non-linear and hard to profile at runtime. Based on their respective characteristics, we design an *analytical* method that directly configures the optimal $\vec{x}$ for CPU clusters, and a *numerical* method that iteratively approaches the optimal $\vec{x}$ for GPU clusters.

3.2 LB-BSP in CPU Clusters

We summarize some typical scenarios where models are trained in CPU clusters without accelerators like GPUs:

(1) Non-neural-network Model Training. Many traditional ML applications like Support Vector Machines (SVM) or Logistic Regression (LR) demand less computing resources than neural networks. They are usually trained in CPU clusters [12], [23], [54], [55].

(2) Non-urgent Model Training. Non-urgent ML tasks may also be trained in CPU clusters opportunistically with leftover resources [12], [56]. For example, to improve cluster utilization, Facebook trains some peripheral face recognition models with the off-peak portions of CPU servers in the diurnal cycle, where the CPU resources would otherwise be wasted [12].

We next present the techniques to fully exploit available resources in such non-dedicated CPU clusters to attain the best model training efficiency.

3.2.1 Performance Characterization of CPU Workers

Static Characteristics. To solve problem (3) for CPU clusters, we first characterize the static properties of CPU workers by measuring their performance against different batch size configurations when there is no resource contention.

(1) Negligible Communication Time: $t^p \gg t^m$ ($t \approx t^p$). Model training is a compute-intensive task for CPU workers, and with built-in optimizations [36] of modern ML frameworks, communication can be hidden by the long computation time (t^p). In our measurements on EC2, when training the Inception-V3 model on ImageNet dataset (introduced later in Section 5.1) with 32 workers and one separate PS (`c5.2xlarge` instances with $\leq 10Gbps$ bandwidth), t^p takes more than 99 percent of the batch processing time t.

(2) Linear Relationship: $\Gamma(x) = x/v$. As batch size quantifies the iteration load, for CPU processors the computation time t^p is *proportional* to the batch size x. To confirm that, we respectively train the ResNet-32 and Inception-V3 model with different types of EC2 instances. We vary x and record the corresponded computation time t^p in Fig. 1, which in each case exhibits strong linearity between x and t^p. Let v be the ratio of x to t^p, i.e., the *sample processing speed*, we then have $t \approx t^p = \Gamma(x) = x/v$.

Given the characteristics above, we solve optimization problem (3) with $x_i = \frac{v_i}{\sum_{j=1}^{n} v_j} X$. Note that such an analytical method essentially merges straggler detection and elimination together. In a nutshell, to set the worker batch size for an upcoming iteration, we only need to know their sample processing speed in that iteration.

Challenges for LB-BSP in Shared CPU Clusters. Although a CPU worker's batch size can be determined with its sample processing speed, when training with resource contention in shared clusters, that sample processing speed may vary dynamically. Therefore, to load-balance workers with adjustments only at the iteration boundaries, we need to predict their sample processing speed before the start of an iteration. As stragglers in non-dedicated CPU clusters can be deterministic and non-deterministic, an ideal prediction approach should be robust to non-deterministic perturbations to avoid over-reaction or oscillation. It should also react to deterministic resource variations quickly.

3.2.2 Predicting Sample Processing Speed

Potential Approaches. A simple solution is to use the last iteration's speed or the Exponential Moving Average (EMA) speed as the predicted one. However, the former is not robust to temporary perturbations, and the latter cannot react quickly to drastic resource variations.

Speed prediction belongs to a classical research problem—*time series prediction*, for which many *statistical* or *learning-based* techniques have been proposed. As a statistical approach, *Autoregressive Integrated Moving Average* (ARIMA) [58] makes predictions with statistics like average and deviation. Meanwhile, models based on *Recurrent Neural Network* (RNN) [59] (like plain RNN and LSTM [60]), with the ability to maintain inner memory, have been applied in forecasting real-world time series like stock price [61] or transport flow [62].

However, the prediction performance of all the above approaches is limited by their *blindness* to the underlying resources. In fact, variations of the driving resources like CPU and memory[3] closely relate to the instantaneous worker processing capability.

This is supported by Fig. 2, in which the model training processes are slowed down after we restrict their resource usage. Such driving resources can help to distinguish the deterministic straggling factors from random perturbations

3. Other resources (e.g., disk I/O, heat) may also impact sample processing speed. Yet instead of exhaustively exploring all the potential impact factors, our goal here is to show the benefits of including the driving resources in prediction, and CPU and memory are two easy-to-measure factors for that.

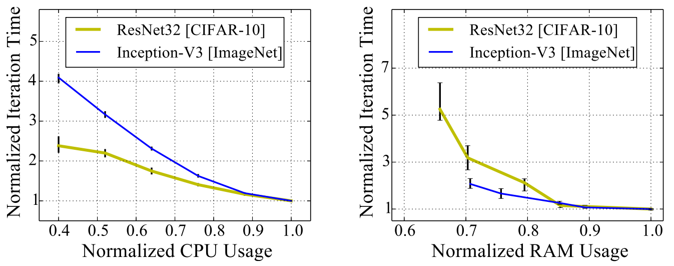


Fig. 2. Sample processing speed is affected by CPU or memory resources. Measurements are conducted with the `stress-ng` tool [57] on a `c5.2xlarge` instance (with swap spaces enabled). The resource usage and iteration time are normalized by the monopolizing case. Each point is an average of 100 iterations.

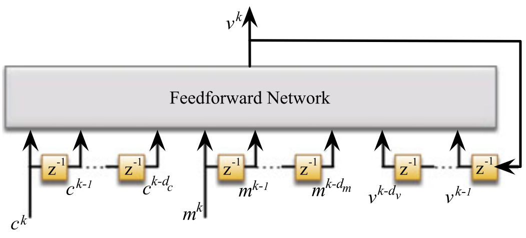


Fig. 3. NARX architecture.

and make more accurate prediction. To this end, we find that *Nonlinear AutoRegressive eXogenous* (NARX) model [28], [63] is a good fit for the prediction of sample processing speed.

NARX Approach. The NARX model we use is basically an extended recurrent neural network that takes three series as inputs: past values of sample processing speed ($\boldsymbol{v}$), current and past values of the two *driving* resources—CPU and memory usage ($\boldsymbol{c}/\boldsymbol{m}$). Essentially, NARX aims to learn a *nonlinear function* $F(\cdot)$ between the predicted speed and a limited view (specified with a *look-back window size*) of the input series

$$v^k = F(v^{k-1}, \ldots, v^{k-d_v}, c^k, \ldots, c^{k-d_c}, m^k, \ldots, m^{k-d_m}). \tag{4}$$

Here v^k, c^k and m^k represent the value of *speed*, *CPU* and *memory* usage in iteration k, respectively; d_v, d_c and d_m represent the corresponded *look-back window size* of each series. Fig. 3 shows the unfolded architecture of the NARX model, in which the input series are fed into a *feedforward* neural network.

The batch size adjusting process with NARX is elaborated in Algorithm 1. In practice, to ensure high prediction accuracy, we maintain a NARX model for each worker, which is trained with the historical execution information (i.e., $\boldsymbol{v}$, $\boldsymbol{c}$ and $\boldsymbol{m}$). To avoid high model complexity, as in existing prediction works [64], [65], the *look-back window* sizes for all the three input series are set to 2, and we include only *one* hidden layer in the feedforward network. Such a simple model can avoid over-fitting and converge fast. Our later evaluation (Section 5.4.1) confirms that such a NARX-based approach can make better predictions than other approaches.

3.3 LB-BSP in GPU Clusters

With strong parallel processing capability, GPUs are the workhorse hardware for training deep neural networks [10], [36], [37]. As elaborated in Section 2.1, neural network models may be trained in non-dedicated GPU clusters: with heterogeneous GPU instances from the spot markets [9], or in

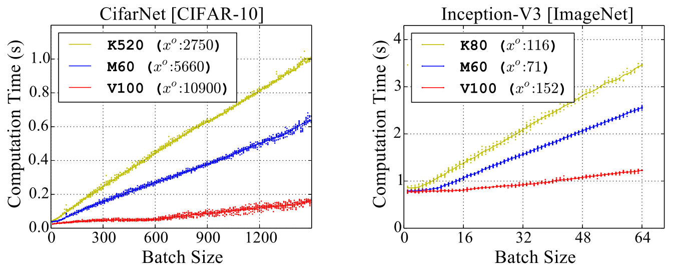


Fig. 4. The relationship between computation time and batch size of different GPU types.

shared GPU clusters [10], [11], [29], [30] where workers may inter-connect at different locality levels, under asymmetric topologies, or be migrated across machines for resource consolidation. Nonetheless, implementing LB-BSP in non-dedicated GPU clusters renders a challenge different with that in CPU clusters. In this part, we first characterize the performance of standalone GPU workers, and then present our LB-BSP algorithm for GPU clusters.

3.3.1 Performance Characterization of GPU Workers

Static Characteristics. To solve problem (3) for GPU clusters, we first profile the static properties of GPU workers. For representativeness, our profiling is conducted with a small neural network—CifarNet [66], and a large neural network—Inception-V3 (their descriptions deferred to Section 5.1).

(1) Non-negligible Communication Time: $t^p \not\gg t^m$ ($t \not\approx t^p$). Computations on GPU workers are usually orders of magnitude faster than CPU workers. Therefore, the communication time in each iteration is no longer negligible when compared with the computation time [36].

(2) Non-negligible GPU Launching Overhead: $\Gamma(0) > 0$. Fig. 4 shows the relationship[4] $\Gamma(\cdot)$ between t^p and x for different GPU types. Even for a very small batch, we find that the GPU computation time could still be considerable. This is because a GPU needs to do a series of preparation work [67] to process each batch (e.g., exchanging parameters between GPU and CPU memory, launching processing kernels), which incurs considerable time overhead regardless of the batch size, and that overhead is particularly salient for large models like Inception-V3. In particular, the existence of GPU launching overhead confirms that *sequential sample processing is highly inefficient*: it is unacceptable to sustain such an overhead when processing each sample.

(3) GPU Saturation Effect: $\Gamma(x) = C, \forall x < x^s$. For advanced GPUs like Tesla V100, the batch computation time t^p would be almost a *constant* if the sample batch is too small (less than the saturation threshold x^s) to *saturate* all the processing kernels [68], [69]. Reducing batch size under x^s does not help to reduce the computation time and would cause GPU underutilization.

(4) GPU Memory Limitation: $x < x^o$. During each training iteration, the input *sample batch*, *model parameters* and *intermediate results* shall all reside in GPU memory. Therefore, the GPU memory size imposes a limit x^o on the maximum

4. To eliminate network interference, the curves in Fig. 4 are obtained with a TensorFlow worker process and a collocated PS process.

batch size (as marked in the legends of Fig. 4), which, to avoid OOM (out-of-memory) error, must be complied with when adjusting the batch size of a GPU worker.

Algorithm 1. Batch Size Updating in CPU Clusters

```
Input: $\{x_i^{k-1}\}$, $\{t_i^{k-1}\}$, $\{c_i^k\}$, $\{m_i^k\}$    ▷ last batch size, last batch
    processing time, current cpu & memory usage of all the
    workers ($i$=1,2,...,$n$)
Require: past values of $\{v_i\}, \{c_i\}, \{m_i\}, i = 1, 2, \ldots, n$;
    $F_i(\cdot), i = 1, 2, \ldots, n$.    ▷ NARX model for each worker
1: procedure CPU_UPDATEBATCHSIZE($k$)
2:   $v_i^{k-1} \leftarrow x_i^{k-1}/t_i^{k-1}, i=1, 2, \ldots, n$.
3:   $v_i^k = F_i(v_i^{k-1}, \ldots, v_i^{k-d_v}, c_i^k, \ldots, c_i^{k-d_c}, m_i^k, \ldots, m_i^{k-d_m}), i=1, \ldots, n$.
4:   $X \leftarrow \sum_{i=1}^n x_i^{k-1}$
5:   $x_i^k = \frac{v_i^k}{\sum_{j=1}^n v_j^k} \cdot X, i = 1, 2, \ldots, n$.
6:   round $x_i^k (i = 1, \ldots, n)$ to integers with $\sum_{i=1}^n x_i^k = X$.
7:   return $\{x_i^k\}$
```

Challenges for LB-BSP in GPU Clusters. The challenges for realizing LB-BSP in GPU clusters are quite different from that in CPU clusters. First, fine-grained GPU sharing is quite rare (due to the technical difficulty and overhead [10], [70]), and auxiliary resources (e.g., CPU, memory, network connectivity) provisioned in GPU clusters fluctuate less often than in CPU clusters [10], [11], [29], [30], leaving it unnecessary to make real-time performance predictions for GPU workers. Second, however, statically profiling the non-linear relationship $\Gamma(\cdot)$ incurs non-trivial programming and time overhead, and is particularly inconvenient for shared GPU clusters where the workers of a ML job may be migrated from time to time. Third, Fig. 4 implies that batch processing time increases *monotonically* with the batch size; meanwhile, compared with the huge number of seconds-level short iterations in the long training process, the occurrence of job migration is much fewer, suggesting that the worker performance is stable in most consecutive iterations.

Therefore, instead of analytically solving problem (3) based on static profiling, for GPU clusters it is more appropriate to employ a *numerical approximation* method.

3.3.2 A Drop-in Algorithm for LB-BSP in GPU Clusters

In this part, we devise a drop-in algorithm to iteratively approximate the *equilibrium* where the gap among all the workers' batch processing time is minimized.

The whole batch size adjusting algorithm is elaborated in Algorithm 2. After each training iteration, we identify two GPU workers: a *leader*—the GPU worker with the *shortest* batch processing time, and a *straggler*—the GPU worker with the *longest* batch processing time. If the *leader* has consecutively preceded the *straggler* during an *observation window* (observation window is introduced for robustness to non-deterministic random variations), we respectively increase (reduce) the *leader* (*straggler*)'s batch size by a certain amount called *step size*. Note that any GPU worker with less than 5 percent available memory would, to avoid OOM error, be excluded from being identified as the *leader*. Moreover, if with Algorithm 2 a worker's batch size is to be reduced below 0, we should restart the training process without that worker, and the resultant setup is actually more efficient.

Moreover, to reduce resource wastage, the equilibrium should be approached efficiently with minimum oscillation. To this end, we introduce two phases: a *fast-approach* phase and a *fine-tune* phase. Initially the algorithm enters the fast-approach phase, where we set a relatively large step size and short observation window; then, once oscillation—a former *leader* now identified as a *straggler* or vice versa—is detected, we switch to the fine-tune phase by reducing the step size (e.g., to 1) and increasing the observation window size.

Algorithm 2. Batch Size Updating in GPU Clusters

```
Input: $\{x_i^{k-1}\}$, $\{t_i^{k-1}\}$, $\{m_i^k\}$ ▷ last batch size, last batch processing
    time & current GPU memory usage of all workers
    ($i$=1,...,$n$)
Require: past values of $\{t_i\}, i = 1, 2, \ldots n$;
    $\Delta \leftarrow 5, D \leftarrow 5$    ▷ $\Delta$: step size; $D$: observation window
    size
1: procedure GPU_UPDATEBATCHSIZE($k$)
2:   leader $\leftarrow \arg\min_i\{t_i^{k-1} \mid m_i^k \leq 0.95\}$
3:   straggler $\leftarrow \arg\max_i\{t_i^{k-1}\}$
4:   $x_i^k \leftarrow x_i^{k-1}, i = 1, 2, \ldots, n$
5:   if $x_{straggler}^{k-1} \leq \Delta$ then
6:     return $\{x_i^k\}$    ▷ print warning: remove this straggler!
7:   if $\forall h \in \{k-D, \ldots, k-1\}, t_{leader}^h < t_{straggler}^h$ then
8:     $x_{leader}^k \leftarrow x_{leader}^{k-1} + \Delta$ ; $x_{straggler}^k \leftarrow x_{straggler}^{k-1} - \Delta$
9:   else if $\exists h \in \{1, \ldots, k-1\}, t_{leader}^h > t_{straggler}^h$ then
10:    $\Delta \leftarrow 1, D \leftarrow 20$    ▷ switch to fine-tune phase
   return $\{x_i^k\}$
```

3.4 Addressing Inconsistent Batch Sizes Under LB-BSP

By tuning the worker batch size with Algorithms 1 and 2, we can effectively eliminate deterministic stragglers and achieve high hardware efficiency. Yet, the resultant batch sizes on different workers would be inconsistent, which may affect the statistical efficiency. In this part, we first elaborate that problem and then give our solution.

Problem of Naive Aggregation Under LB-BSP. Under BSP, the aggregated global gradient g for parameter updating is the naive average of the gradients from all the workers. Suppose there are n workers and g_i is the gradient calculated on worker-i ($i = 1, 2, \ldots, n$), then

$$g = \frac{1}{n}\sum_{i=1}^{n} g_i, \text{ where } g_i = \frac{1}{|B_i|}\sum_{s\in B_i} \nabla l(s, \omega). \tag{5}$$

Here B_i is the batch on worker-i. Getting rid of g_i, we have

$$g = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{|B_i|}\sum_{s\in B_i}\nabla l(s, \omega) = \sum_{i=1}^{n}\sum_{s\in B_i}\frac{1}{n|B_i|}\nabla l(s, \omega). \tag{6}$$

This implies that, when workers have different batch sizes ($|B_i|$), the *significance* of different samples, i.e., $\frac{1}{n|B_i|}$, is also different. Thus g is biased to samples in small batches, which may harm the statistical efficiency. To verify that, we train the Inception-V3 model under Algorithm 2 in a 16-node heterogeneous GPU cluster (i.e., Cluster-A in

Section 5.1). After traversing the ImageNet dataset for 20 epochs, the training accuracy only reaches 43 percent, much worse than that under BSP (59 percent).

Weighted Gradient Aggregation. To avoid biased gradient, we propose *weighted gradient aggregation*—using a worker's batch size as the weight when aggregating gradients. Suppose the total batch size is $\sum_{j=1}^{n}|B_j|=X$, then we have

$$g=\frac{1}{\sum_{j=1}^{n}|B_j|}\sum_{i=1}^{n}|B_i|\cdot g_i=\sum_{i=1}^{n}\sum_{s\in B_i}\frac{1}{X}\cdot\nabla l(s,\omega). \tag{7}$$

Obviously, now each sample plays an equal role in parameters updating, regardless of the batch size inconsistency. After that fix, for *i.i.d.* dataset, i.e., samples being *independent* and *identically distributed*, LB-BSP can achieve identical statistical efficiency with BSP, which is known to be optimal.

3.5 LB-BSP for Cross-Silo Federated Learning

So far, we focus on scenarios that the ML practitioners have full control over the training datasets. However, in many real-world scenarios, training samples are privacy-sensitive and dispersed on distributed clients like cellphones, banks and hospitals. To train models without centralizing such private data, an increasingly popular technique is federated learning (FL) [32], [33], [34], [45], [71], under which clients can compute model updates locally and have the updated models periodically synchronized on a central server.

Depending on the use cases, FL setups can be broadly classified into *cross-device* FL [32], [45], [71] and *cross-silo* FL [33], [34]. In the cross-device setting, clients are a large number of mobile or IoT devices with limited computing power and unreliable communications [45]. In contrast, clients in cross-silo setting are a small number of organizations with reliable communications and computing resources [33]. Model synchronization is very *expensive* in both setups. For cross-silo FL where the link bandwidth is usually large, since it has much more rigid privacy requirements (e.g., even the server cannot access the model parameters), the model updates must be carefully encrypted (decrypted) before (after) transmission; a typical encryption method is Homomorphic Encryption (HE) [72], which is very time-consuming (inflates the synchronization overhead by about two orders of magnitude [33]). Therefore, for better efficiency, a common training algorithm adopted for FL is *FedAvg* [32], which dictates each client to iterate for *multiple* times before conducting *one* global synchronization. More specifically, when specifying the load in each round (one round corresponds to one synchronization), under FedAvg there are two hyper-parameters[5]: τ and B. Here τ is the number of local iterations, and B is the batch size.

As elaborated in Section 2.1, the FL setup is a typical non-dedicated scenario where faster clients would be slowed down by the straggling ones. This is more of a problem for cross-silo FL. In cross-device FL, there are usually a vast number of clients (e.g., cellphones) that dynamically join or leave the FL process. It is thus very hard, if not impossible, to keep the execution status of each client. And in fact, given client abundance, waiting for updates of all the clients is unnecessary under state-of-the-art cross-device FL systems [45]. In contrast, in cross-silo FL the participating clients are limited and long-living, making it necessary and also feasible to mitigate the stragglers within. Therefore, our LB-BSP design in this part focuses on cross-silo FL.

Challenges for LB-BSP in Cross-Silo FL. Compared with in shared CPU clusters (Section 3.2), client resources for cross-silo FL are usually much more stable [33], making the NARX prediction method no longer necessary. With a simple EMA method, we can easily sense the client processing capability[6] and obtain their load tuning scale in a way similar to Section 3.2. Nonetheless, under the distinct FedAvg algorithm where both τ and B determine the per-round load of a client, a first question for our LB-BSP design is how to enforce load scaling—on τ or B? And the second question is, since under privacy constraint the local datasets on different FL clients are *non-i.i.d.* and *cannot* be migrated, how to ensure convergence validity after load tuning?

Regarding the first question, we propose to enforce load scaling by tuning B instead of τ. Essentially, by introducing τ, the FedAvg algorithm allows each client to proceed in its local optimal direction (which is different among clients due to *non-i.i.d.* datasets) for multiple steps in each round. If we adjust τ based on client processing speed, then the faster clients can proceed for more steps and make the aggregated updates biased. Therefore, we choose to preserve τ and have the load tuning enforced by B.

Regarding the second question, we first learn that weighted gradient aggregation (WGA) is no longer appropriate for FL. A basic assumption of WGA is that the datasets are *i.i.d.* or can be migrated (so that each sample as well as each sample class can play an equal role in model refining, as explained in Section 3.4), which does not hold for FL. If we adopt WGA and scale the learning rate η based on B, then since faster clients process more samples in each round, their local datasets gain a bigger say in model training. Therefore, for FL scenarios, it is more important to ensure that each local dataset instead of each individual sample is equally treated, and we shall thus preserve the learning rate η on each client regardless their batch size B. Since samples on each client can be shuffled *locally*, the *expected* gradient from each sample batch *stays the same* even with a different batch size. This way, we can in fact prevent clients from cheating by making faster computations.

To summarize, to realize LB-BSP for cross-silo FL, we scale B in a similar way as in shared CPU clusters (except that we adopt EMA instead of NARX as the prediction method), with the learning rate η on each client preserved. We will evaluate such design in our EC2 experiments (Section 5.5).

5. An equivalent representation form of τ is E—the number of training pases each client makes over its local dataset in each round [32]. Given the local dataset size d, we have $E*d=\tau*B$.

6. In cross-silo FL, the synchronization overhead is mainly incurred by encryption/decryption, and the performance bottleneck lies mostly in computation instead of pure communication [33]. Thus, for simplicity we skip the pure communication time here as in Section 3.2. If in some extreme cases the communication time needs to be considered, we can explicitly measure it and extend the load-tuning algorithm in LB-BSP to counteract any client inconsistency in such communication time.

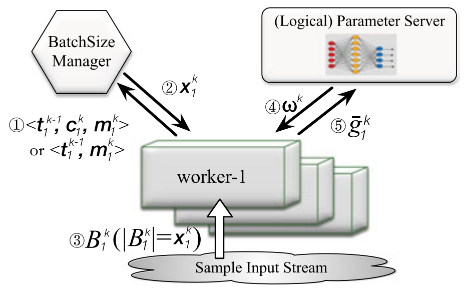


Fig. 5. LB-BSP workflow in iteration k (the circled numbers represent the execution order). The logical PS can be in the form of distributed shards, or be replaced by the `All-Reduce` architecture.

4 Implementation

We implement LB-BSP in `BatchSizeManager`,[7] a Python module that can be integrated into various ML frameworks, including TensorFlow [8], PyTorch [26] and MXNet [25].

Algorithm 3. LB-BSP Workflow

```
Worker: i=1, 2, ..., n:
1:  procedure WORKERITERATE($k$)
2:    $x_i \leftarrow$ BatchSizeManager. UpdateBatchSize($<$ states $>$)
          ▷ blocking in CPU clusters and non-blocking in GPU
          clusters
3:    load the next data batch $B_i^k$ such that $|B_i^k| = x_i$
4:    pull $w^k$ from PS
5:    calculate local gradient $g_i^k$
6:    push $\bar{g}_i^k \leftarrow \frac{nx_i}{X} g_i^k$ to PS ▷ $\bar{g}_i^k$: weighted gradient
Parameter Server (PS):
1:  procedure PARAMETERSERVERITERATE($k$)
2:    aggregate gradient $g^k \leftarrow \frac{1}{n}\sum_{i=1}^n \bar{g}_i^k = \frac{1}{X}\sum_{i=1}^n |B_i^k| g_i^k$
3:    update parameters $w^{k+1} \leftarrow w^k - \eta g^k$ ▷ $\eta$: learning rate
BatchSizeManager:
1:  procedure UPDATEBATCHSIZE $<$ states $>$
2:    Refer to Algorithm 1 (CPU cluster) or Algorithm 2 (GPU
      cluster).
```

Architecture Overview. The overall workflow of LB-BSP is described in Algorithm 3 and Fig. 5. In the beginning of iteration k, each worker pushes its latest execution information ($\langle$ batch processing time t^{k-1}, CPU usage c^k, memory usage m^k $\rangle$ for CPU clusters, or $\langle t^{k-1}, m^k \rangle$ for GPU clusters) to the `BatchSizeManager`, and then pulls back the updated batch size x^k. Note that LB-BSP is also applicable if the gradients are aggregated with the All-Reduce architecture.

In particular, for CPU clusters the batch size updating process is *blocking* so that the `BatchSizeManager` can get the latest state information for speed prediction; yet for GPU clusters, it is *non-blocking* because GPU workers' state is more stable (Section 3.3), and non-blocking interaction can avoid prolonging the very short GPU iterations. To that end, on each GPU worker we launch a separate thread to update batch size in the background.

Enabling Variable Batch Size. In existing ML frameworks, batch size is set as a constant when defining the computation graph, with no direct APIs to configure it during the training process. To enable variable batch size, in TensorFlow we decouple the batch size from the symbolic dataflow graph, and specify it as a `tensor` value passed to TensorFlow `session` through the `feed_dict` API. For MXNet and PyTorch, we respectively customize the `DataIter` and `BatchSampler`so that they can accept a user-specified batch size at runtime and generate a corresponded sample batch.

Measuring Worker Execution Status. To implement LB-BSP, at the worker side, we need to obtain the batch processing time t and state information (e.g., CPU or memory usage). Acquiring t is easy in *dynamic-graph* based frameworks like PyTorch, but is challenging for *static-graph* based frameworks like TensorFlow or MXNet. In TensorFlow, each iteration (forward or backward propagation and synchronization) is executed as a whole with `tf.Session()`, which cannot be decomposed with simple instructions. We choose to profile the batch processing time from the `Timeline` log once each iteration finishes. Instead of respectively measuring the communication time and computation time which might overlap with each other, we directly calculate batch processing time as the *iteration time* minus the *synchronization waiting time* (i.e., duration of the `sync_token_q_Dequeue` operation). Besides, to measure the CPU or memory usage we resort to the Python `psutil` [73] library, and to measure the GPU memory usage we adopt the `tf.contrib.memory_stats.BytesInUse()` operation.

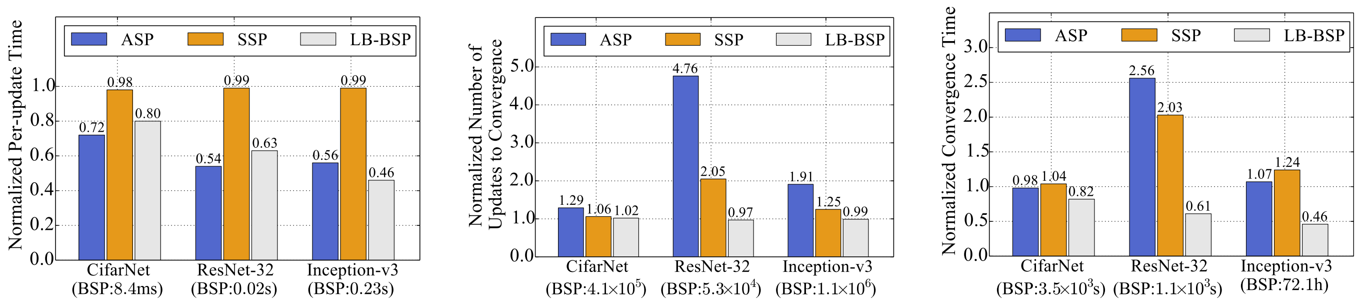
Thrift RPC Protocol. The `BatchSizeManager` can be located in a dedicated machine or co-located with the PS process on an existing server of the cluster. For efficient communication between the `BatchSizeManager` and workers, we employ Apache Thrift [74], a lightweight RPC protocol developed by Facebook, and we create a thread pool to serve the worker requests in parallel.

Online NARX Training. The NARX models we use for CPU clusters are written in Keras [75], a high-level neural network API. Accurate NARX training requires collecting enough samples, so we enable NARX prediction only after the first 500 iterations. In practice we find that 500 samples are enough for accurately training our NARX models, which are quite simple (Section 3.2.2). Within the first 500 iterations, we can use EMA or, if that training job is recurring, the past NARX models trained in former runs. For fast convergence, we initialize NARX models by *model reusing* [76]—with the models trained even for other workers or ML jobs. Besides, we also adopt *early stopping* [77] and in practice we found that most training processes terminate within 10 steps.

5 Evaluation

In this section we systematically evaluate the performance of LB-BSP in non-dedicated clusters. We start with the end-to-end (Section 5.1) and micro-benchmark (Section 5.2) evaluations in GPU clusters. Then we resort to model training with leftover resources in production CPU clusters (Section 5.3), with a deep dive analysis of the NARX prediction approach (Section 5.4). Further in Section 5.5 we demonstrate LB-BSP effectiveness in cross-silo FL, and finally we

7. While LB-BSP can be integrated into the engines of ML frameworks, this would lose generality, mess the decoupled programming logic of input and graph propagation modules, and also lose the flexibility to switch to All-Reduce communication backend (Operations in Algorithms 1 and 2 are hard to be realized with All-Reduce semantics).



(a) Hardware Efficiency (b) Statistical Efficiency (c) Overall Convergence Efficiency

Fig. 6. Training efficiency under different worker coordination schemes in Cluster-A.

evaluate the overhead of our LB-BSP implementation in Section 5.6.

5.1 End-to-End Evaluations in GPU Cluster

Experimental Setup. As elaborated in Section 2.1, there do exist some scenarios (e.g., due to budget limitation or fairness policy) where model training has to be conducted in heterogeneous GPU clusters. To emulate such scenarios, we build *Cluster-A*, a heterogeneous GPU cluster with 16 Amazon EC2 instances: four `p3.2xlarge` instances (each with one Tesla V100 GPU), four `g3.4xlarge` instances (each with one Tesla M60 GPU), four `p2.xlarge` instances (each with one Tesla K80 GPU), and four `g2.2xlarge` instances (each with one GRID K520 GPU). The link bandwidth of each GPU instance is over 10Gbps. On each instance we run a worker process and a collocated PS shard under TensorFlow.

With Cluster-A, we train the CifarNet [66] and ResNet-32 [78] model on CIFAR-10 dataset (which contains 32×32 colored images of 10 classes, with 50K images for training and 10K for testing), and Inception-V3 [79] model on ImageNet dataset [80] (containing 1.28 million training images of 1000 classes). CifarNet is a small convolutional neural network [66] designed for CIFAR-10 dataset; ResNet-32 [78] is a middle-size neural network with residual connections cross different layers, and Inception-V3 [79] is a large neural network improved from GoogLeNet. For simplicity each worker locally hosts a full dataset copy. The initial batch size of each worker is set to 128 for CIFAR-10, and 32 for ImageNet. The initial learning rate is set to 0.01.

The schemes evaluated in this part are BSP, ASP, SSP[8] and LB-BSP, and we defer the comparisons with FlexRR and redundant execution to CPU clusters.[9] We measure the overall training efficiency as well as the hardware and statistical efficiency. The results are summarized in Fig. 6, where BSP is the baseline and all the values displayed are normalized by that under BSP.

Hardware Efficiency. The metric we use for hardware efficiency is *per-update time*—the average time it takes for PS to receive one gradient update. For BSP and LB-BSP, per-update time is the average iteration time divided by the number of workers. In Fig. 6a, LB-BSP remarkably outperforms BSP and SSP, and this is consistent with our analysis in Section 2.2. Interestingly, LB-BSP is even 15 percent better than ASP in hardware efficiency when training Inception-V3 —we will explain that with micro-benchmark evaluations in Section 5.2.

Statistical Efficiency. Statistical efficiency is measured as the number of updates required to reach the target accuracy. We set different near-optimal accuracy targets for different models: 0.80 for CifarNet, 0.86 for ResNet-32, and 0.65 for Inception-V3. As shown in Fig. 6b, for each model under LB-BSP, the number of updates required to reach the target accuracy is almost identical with that under BSP. In contrast, ASP and SSP require up to $4.76\times$ and $2.05\times$ [10] the number of BSP to make that accuracy.

Moreover, ASP and SSP fall behind not only in the convergence speed, but also in the the ultimate accuracy attained. Fig. 7 shows the convergence curves of ResNet-32 and Inception-V3, where an epoch is a full pass of all the samples in the CIFAR-10 or ImageNet dataset. For ResNet-32, we find that the accuracy made by ASP or SSP gets stuck below 0.88, while BSP and LB-BSP successfully make the ideal accuracy of 0.92. Also, for Inception-V3, BSP and LB-BSP already surpass ASP and SSP even for reaching a sub-optimal accuracy target 0.65.[11]

Overall Convergence Efficiency. Fig. 6c shows the overall time required to reach the target accuracy, where LB-BSP surpasses the second best by up to 54 percent. Therefore, it's highly rewarding to employ LB-BSP in such heterogeneous GPU clusters. We further train ResNet-32 in a 12-node GPU cluster without the `p3.2xlarge` instances, and the iteration speedup of LB-BSP over BSP reduces (from 37 percent in Fig. 6a) to 28 percent. Thus, the more heterogeneous the cluster is, the more necessary it is to adopt LB-BSP for load balancing.

5.2 Micro-Benchmark Evaluations in GPU Clusters

5.2.1 LB-BSP Behavior Deep Dive

To further understand LB-BSP behavior from the micro level, we scale down Cluster-A to contain only 4 GPU

8. By default, TensorFlow does not support SSP. We implemented it with a `worker-coordinator` module over the Thrift RPC protocol, which enforces fast workers to wait if the slowest one is 5 iterations behind.

9. We exclude FlexRR from GPU evaluations because it is not compatible with the tensor-based processing style of GPUs, and redundant execution is also excluded because its behavior in heterogeneous GPU clusters is trivial—always ignoring the most inferior GPU worker(s).

10. Being a deep model with residual connections, ResNet-32 is more sensitive to parameter staleness and the performance degradation of ResNet-32 under ASP and SSP is more salient than CifarNet or Inception-V3.

11. Inception-V3 is not trained to the ideal accuracy (78.8 percent) due to our budget limitation. For reference an existing work [50] has confirmed that BSP can yield a higher final accuracy than ASP even in homogeneous clusters.

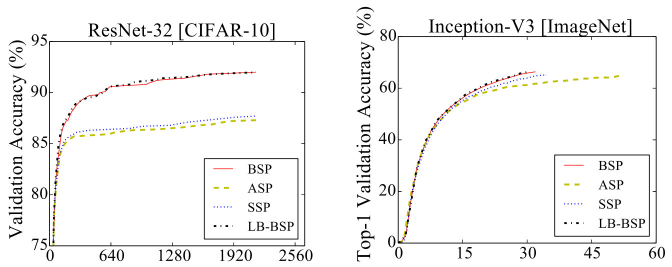


Fig. 7. Test accuracy against training epochs.

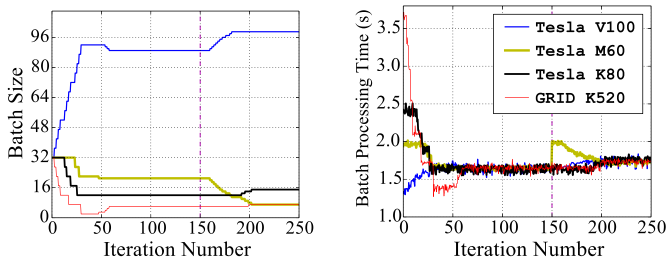


Fig. 8. Instantaneous batch size and batch processing time of the four heterogeneous GPU workers under LB-BSP. At iteration 30, Algorithm 2 enters the fine-tune phase, and batch sizes of the four workers stabilize at (89,21,12,6). Later at iteration 150, bandwidth of the M60 GPU worker is reduced to emulate a migration-caused locality degradation, and the four workers then enter a new equilibrium.

instances each of a distinct type, and measure the instantaneous worker batch size and batch processing time when training Inception-V3. As shown in Fig. 8, LB-BSP gradually increases the batch size of the most powerful worker (i.e., the one with the Tesla V100 GPU), with the batch sizes of the other workers correspondingly decreased. Finally an equilibrium is reached where all the workers share almost the same batch processing time. Note that after iteration 30, the LB-BSP algorithm (Algorithm 2) enters the fine-tune phase, where the batch sizes of the four workers gradually stabilize at (89,21,12,6).

Fig. 8 also helps to elaborate why LB-BSP can even outperform ASP in hardware efficiency. By yielding 26 samples (from 32 to 6), the worker with K520 GPU has its batch processing time reduced by nearly 2s; in contrast, the worker with V100 GPU, after incorporating as many as 57 samples, only suffers an increase of 0.5s. Therefore, by allowing advanced workers to process more samples and achieve a higher utilization with negligible slowdown, LB-BSP can reduce the average cost to process one sample and improve the hardware efficiency.[12]

Furthermore, we evaluate LB-BSP applicability in shared GPU clusters, and emulate the network variation caused by locality degradation when conducting cross-machine(rack) worker migration. In Fig. 8, at iteration 150 we bound the bandwidth of the worker with M60 GPU to 2.5Gbps (from EC2-provisioned 10Gbps, with the `wondershaper` [81] tool). The LB-BSP algorithm learns that shortly and then gradually adjusts workers' batch size to reach another equilibrium. This confirms that LB-BSP can work well in multi-tenant GPU clusters with job migrations from time to time.

12. This also holds for CifarNet and ResNet-32 in Fig. 6a, but their iterations are shorter and the GPU random perturbations are more significant, rendering ASP still better than LB-BSP in hardware efficiency.

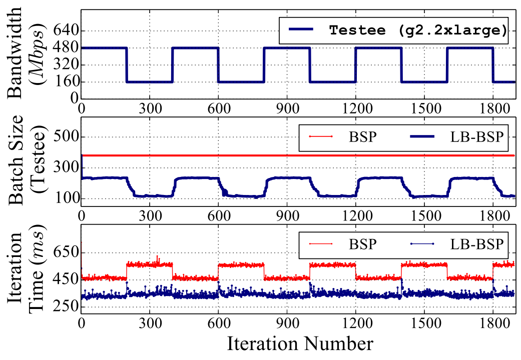


Fig. 9. Instantaneous batch size and iteration time of the testee-worker (`g2.2xlarge`) when oscillating its bandwidth respectively under BSP and LB-BSP.

5.2.2 LB-BSP Behavior Facing Oscillating Bandwidth

To further reveal LB-BSP effectiveness in shared GPU clusters with periodically network variations, we build a small cluster with oscillating network bandwidth. That cluster contains 8 single-GPU workers: four `g2.2xlarge` instances, two `p2.xlarge` instances, and two `g3.4xlarge` instances. We train the ResNet-32 model on CIFAR-10 dataset, and for generality we switch to the MXNet framework in this experiment and the initial batch size is 380. Moreover, we select one `g2.2x` worker as the testee, and periodically rotate its link bandwidth (also with the `wondershaper` tool) between *abundance* state ($480 Mbps$) and *deficit* state ($160 Mbps$). For performance comparison between LB-BSP and BSP, we record the testee's instantaneous batch size and the corresponded iteration time, as shown in Fig. 9.

From Fig. 9, we can learn that LB-BSP outperforms BSP in two aspects. First, facing the *static* hardware (GPU) heterogeneity, LB-BSP automatically sets each worker's batch size based on their computing capabilities: batch size of the `g2.2xlarge` instance is reduced from 380 to 235 to compensate for hardware deficiency. This brings a performance improvement of around 27 percent. Second, facing the *dynamic* resource variation (i.e., bandwidth oscillation of the testee worker), LB-BSP can swiftly perceive the testee's prolonged batch processing time and adjust its batch size as a remediation. After the testee's bandwidth drops from 480 Mbps to 160Mbps, the iteration time under LB-BSP increases by less than 3 percent. As a result, compared with BSP, LB-BSP improves the hardware efficiency by over 41 percent.

5.3 End-to-End Evaluations in Shared CPU Cluster

As elaborated in Section 3.2, non-neural-network or not-so-urgent models may be trained in non-dedicated CPU clusters. In this part we systematically evaluate LB-BSP performance in such CPU clusters.

Experimental Setup. We manually created *Cluster-B*, a heterogeneous CPU cluster that emulates the shared production environment where ML models are trained with the dynamic leftover resources [12], [27], [40], [82]. Cluster-B is built based on a one-hour snapshot of Google Trace [27]. That trace discloses the machine configurations (CPU/Memory capacities in *normalized* form) of a production cluster from Google, together with the information of all the

TABLE 2
Cluster-B Composition

Instance Type	CPU, Mem (core, GiB)	Num
m4.2xlarge	(8, 32)	17
c5.2xlarge	(8, 16)	10
r4.2xlarge	(8, 61)	2
m4.4xlarge	(16, 64)	2
m4.xlarge	(4, 16)	1

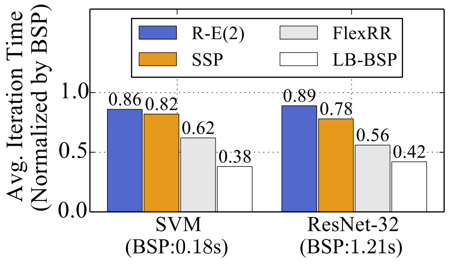


Fig. 10. Iteration time with different schemes in Cluster-B.

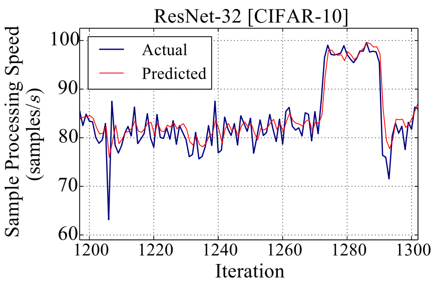


Fig. 11. NARX prediction result on an m4.2xlarge instance.

involved jobs/tasks during a selected month—including their resource consumptions and start/end times.

More specifically, we *scale down* the totally 12,583 machines to 32 EC2 instances, with the former's *hardware heterogeneity proportionally* preserved—by accordingly selecting the instance types and the quantity of each type, as summarized in Table 2. Meanwhile, *resource dynamicity* of that Google cluster is also emulated: we randomly map each instance to a machine in the Google cluster, and launch a set of faked tasks sharing identical behaviors (i.e., start/end times & CPU/memory consumptions) with those submitted to that Google machine.

Regarding the models, we train SVM on a malicious URL dataset [83], and ResNet-32 on CIFAR-10 dataset. The batch size for SVM training is 10 percent of the whole URL dataset (as in [54]), and is 128 for ResNet-32.

The schemes evaluated are Redundant Execution (R-E), SSP, LB-BSP and FlexRR. The first three schemes are implemented in TensorFlow. For R-E, we add two c5.2xlarge instances (also in resource contention with some faked tasks) to Cluster-B as the backup worker, and only collect 32 gradients returned the earliest in each iteration. R-E is supported in TensorFlow with the replicas_to_aggregate parameter. In SSP, the bound of iteration gap is still set to 5, as in Section 5.1. Regarding FlexRR, since it is not open-sourced and its sequential sample processing manner is not supported in current ML frameworks (Section 2.2.3), we choose to emulate it in PyTorch. We generate tiny batches each containing only one sample, and then encapsulate them into logical batches of designated size; such a logical batch can then be processed in a sequential manner. Meanwhile, each worker's helper group is set to be all the remaining workers, and for the other setups (e.g., progress check frequency, trigger condition of load reassignment), we follow the suggested values in [23].

Hardware Efficiency. Fig. 10 shows the average iteration time when training SVM and ResNet-32 under different schemes in Cluster-B. For fair comparison, each value in Fig. 10 is normalized by the BSP performance in the corresponded framework (sequential-processing-style PyTorch for FlexRR, and TensorFlow for the others). As in Fig. 10, R-E and SSP speed up the training iterations only marginally, because they focus on either worst-case or transient stragglers, but the stragglers in Cluster-B span a wide range of degrees and durations. Meanwhile, regarding FlexRR, its performance is better but not the best, because its decentralized load balancing decisions are not optimal, and meanwhile it suffers non-negligible measurement, negotiation and load reassignment overheads (Section 2.2.3). In contrast, LB-BSP can bring a speedup of 62 percent for SVM and 58 percent for ResNet-32 over BSP, outperforming the second best (FlexRR) by up to 38.7 percent. Thus, given that LB-BSP can also make the optimal statistical efficiency (Section 5.1), it can surely lead to the best convergence efficiency among all the schemes evaluated.

5.4 Micro-Benchmark Evaluations in CPU Clusters

In this part, we first evaluate the prediction performance of the NARX model (Section 3.2.2) we use in Cluster-B, and then compare LB-BSP performance in clusters with different heterogeneity levels.

5.4.1 NARX Performance Deep Dive

Visual Analysis. To get a *visual* understanding of the NARX prediction performance, we randomly select a period (iteration 1200∼1300) from the ResNet-32 training process on one m4.2xlarge instance of Cluster-B; the *actual* and *predicted* sample processing speeds are presented in Fig. 11. From it we observe that the benefit of NARX is twofold. On the one hand, NARX is robust to non-deterministic *transient* perturbations: when there are "*spikes*" (like the sharp wave around iteration 1206) in the actual speed curve, the predicted curve fluctuates much less. This is because NARX predicts also with the worker's available memory and CPU amounts, which are relatively stable during those "spikes". On the other hand, when the actual speed increases not for randomness but for non-transient deterministic factors like increased CPU or memory resources (e.g., around iteration 1270), the predicted speed can promptly catch up.

Comparison With Other Approaches. We further compare NARX with other approaches surveyed in Section 3.2.2, as listed in Table 3. The *memoryless* method means to take last iteration's sample processing speed as the predicted one. Regarding the EMA approach, the smoothing factor α (weight of the latest observation) is set to be 0.2. As for the statistical prediction approach—ARIMA, its *order of the autoregressive model* (p), *degree of differencing* (d), and *order of the moving average* (q) are respectively set to 2, 2 and 1, based

TABLE 3
RMSE and Iteration Speedup When Applying Different Prediction Approaches in LB-BSP

Method	Configuration	RMSE	Avg. Iteration Time (Normalized by BSP)
Memoryless	-	11.85	0.58
EMA	α=0.2	7.85	0.48
ARIMA	(p,d,q)=(2,2,1)	9.67	0.52
SimpleRNN	look-back=2	8.34	0.49
LSTM	look-back=2	9.19	0.51
NARX	look-back=2	4.78	0.42

on the model selection techniques [84]. Finally, for SimpleRNN (plain RNN) and LSTM, their *look-back* window size is set to 2, the same as in NARX.

Then, we replace the NARX approach with those candidate prediction approaches, and re-train ResNet-32 model under LB-BSP in Cluster-B. For each approach, we record the average *root-mean-square error* (RMSE) of the prediction results and the corresponded iteration time (*normalized* by the iteration time under BSP). From Table 3, the NARX approach, with the ability to perceive CPU/memory resource variations, attains the best performance—it surpasses the second best by around 40 percent in RMSE and 15 percent in average iteration time.

5.4.2 LB-BSP With Different Straggler Extents

To further evaluate LB-BSP under different heterogeneity extents, we build another CPU cluster with competing processes injected at controlled levels.

Experimental Setup. We first build a homogeneous CPU cluster containing 32 workers, where each worker is a `c5.2xlarge` instance with 8 CPU cores and 16GB memory. Then we inject artificial stragglers by running on each worker a *competing process*; that process consumes designated cpu cycles and memory space. In particular, to emulate resource *dynamicity*, we make that competing process periodically run or sleep with certain probability; to emulate resource *heterogeneity*, in different workers that probability and the resource consumption of the competing process are different. By tunning those configurations, we create three heterogeneity levels: *Homo*—no stragglers at all; *Hetero-L2* (*L3*)—sample processing speed of the slowest worker is roughly 1/2 (1/3) of the fastest one.

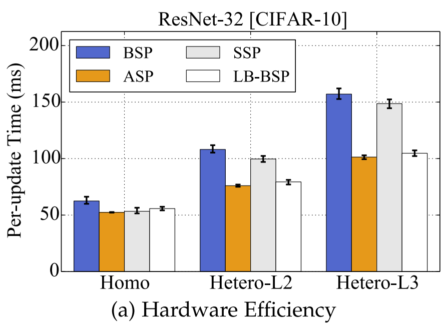
Efficiency Comparison. Fig. 12 shows the performance under different synchronization schemes when training ResNet-32 model on CIFAR-10 dataset. We repeat each training process for 3 times, and report min/max value of each metric via error bars. As revealed by Fig. 12, LB-BSP is the best one among all the schemes, its superiority increasing with the level of heterogeneity. More specifically, in each case, LB-BSP requires the same number of updates for convergence as BSP, and meanwhile shares similar average iteration time with ASP (gap $< 5\%$). This indicates that LB-BSP attains the *best statistical efficiency*and *near-optimal hardware efficiency*. From Fig. 12c, at Hetero-L3, LB-BSP outperforms the other schemes by more than 30 percent.

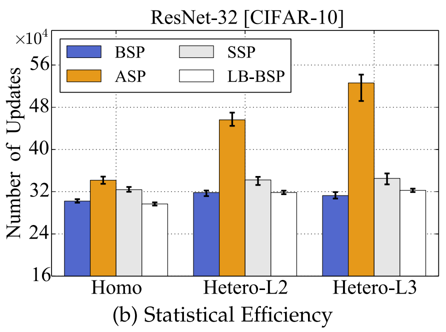
5.5 Evaluations in Cross-Silo Federated Learning

Note that in Section 3.5 we propose our LB-BSP design for cross-silo FL scenarios, with B tuned but learning rate η preserved at its heart. In this part we then evaluate its effectiveness.

Experimental Setup. We set up a 5-client cluster on Amazon EC2 to conduct cross-silo FL, with one `m4.4xlarge` instance, one `c4.2xlarge` instance, one `c5.2xlarge` instance, one `c4.xlarge` instance and one `c5.xlarge` instance. The server is a `c5.4xlarge` instance. The models we train in this cluster are LeNet-5 [85]—a 7- layer simple convolutional neural network suitable for FL scenarios, and a LSTM model with two recurrent layers. The LeNet-5 model is trained on the CIFAR-10 dataset and the LSTM model is trained on KWS dataset—a subset of the Speech Commands Dataset [86] with 10 keywords. As an indicator of the worst case, in our setup the training datasets are extremely *non-i.i.d.* across different clients: we partition the CIFAR-10 or KWS dataset based on the sample labels and have each client host exactly two sample classes. The default values of the local iteration number (τ), batch size (B) and learning rate (η) are 500, 100 and 0.01, respectively.

Efficiency Comparison. With the above setup, we measure the FL training efficiency with and without LB-BSP. Under standard FedAvg algorithm with identical τ, B and η on each client, faster clients would have to wait for slower ones in each synchronization round. Execution results reveal that, in average the fastest client (`m4.4xlarge` instance) spends over one third of the per-round time waiting for the slowest client (`c5.xlarge` instance). In contrast, by adaptively tuning B, our LB-BSP design can remarkably mitigate such resource wastage. Fig. 13 shows the best-ever accuracy during each FL training process, which implies that our LB-BSP design can largely speed up model convergence with negligible accuracy loss. To be specific, for LeNet-5, LB-BSP attains a speedup of over 27 percent, and by preserving η, it makes a near-optimal accuracy performance (0.42) as under standard FedAvg. In contrast, those LB-BSP variants





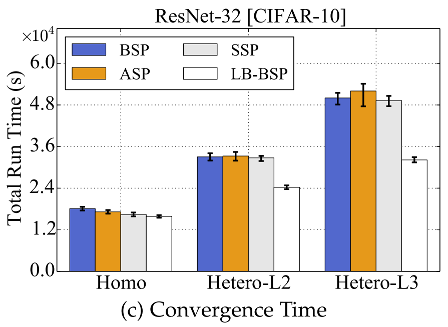


Fig. 12. Efficiency when training ResNet-32 under different straggler extents.

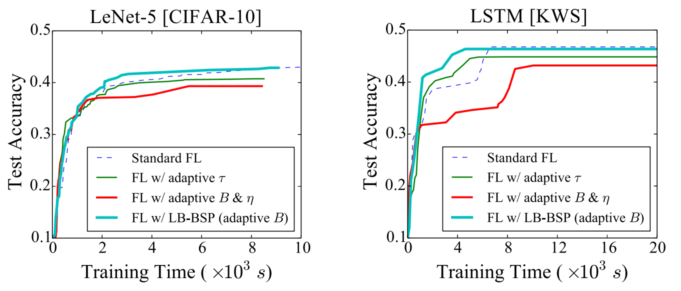


Fig. 13. Test accuracy curves for LeNet-5 and LSTM in cross-silo FL. Our LB-BSP design (with batch size B adaptively adjusted and learning rate η preserved) can remarkably speed up model convergence with accuracy preserved.

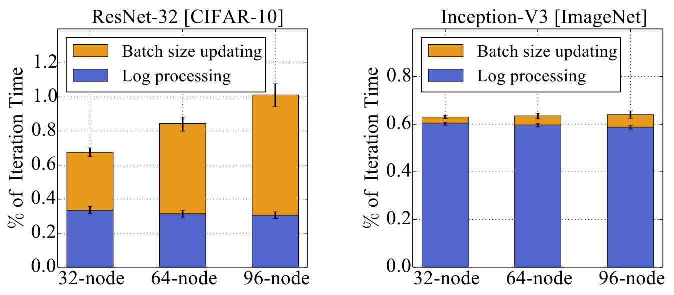


Fig. 14. LB-BSP overheads in CPU clusters.

discussed in Section 3.5—one that tunes τ instead of B and the other that tunes η proportionally with B—both fail to make such an accuracy, because under those schemes the significance of different dataset partitions (i.e., different sample classes) would be inconsistent.

5.6 System Overhead and Scalability

In TensorFlow, LB-BSP introduces two extra procedures: first to extract batch processing time from execution logs (e.g., the `Timeline` object), and second to conduct RPC communication between the `BatchSizeManager` and workers. Yet, they won't slow down GPU workers because of our non-blocking design (Section 4), and here we measure the slowdown caused by those two extra procedures in CPU clusters.

We respectively train ResNet-32 and Inception-V3 model for 1000 iterations in three CPU clusters—a *homogeneous* cluster with 33 `c5.2xlarge` instances (32 worker nodes and 1 separate node hosting both the PS and `BatchSizeManager`), and then its *enlarged* version with a *doubled/tripled* number of workers. Fig. 14 shows the average time respectively spent on log processing and batch size updating, *normalized* by the iteration time (error bars show the 5th/95th percentile). Even in the largest cluster with 96 workers, the total overheads are less than 1.1 percent of the iteration time for both models. This indirectly confirms that, performance of the PS is almost not affected by the co-located `BatchSizeManger`.

6 Additional Related Work

In addition to Section 2.2, in this section we introduce some other works related to this paper. We first review the research works on data batching, and then discuss the efficiency optimization methods on other aspects.

Batching is necessary when processing long-lasting streaming inputs or training models with large datasets. For big data streaming systems [87], [88], some works [89], [90] have explored how to adaptively adjust the batching interval when faced with dynamic data rates or operating conditions. For iterative model training, some [91], [92] have proposed to adaptively increase batch size during the training process to yield faster convergence. Yet, those works don't involve workload allocation among parallel workers, and are thus orthogonal to LB-BSP.

Meanwhile, it is recently a very hot research topic to enhance the training efficiency of distributed ML. While our focus is straggler elimination, on efficiency enhancement there are some other research branches, and a well-known one is improving the communication efficiency. In the literature, communication efficiency can be improved by pipelining the layer-wise parameter transmissions [36], [93] and by optimizing the transmission order of different workers or different parameter groups [14], [94]. Some other works instead proposed to compress the communication content by *quantization*—reducing the bandwidth consumption by transferring the low-precision gradients [95], [96], or by *sparsification*—sending only a portion of the updates that are significantly enough [97], [98]. Additionally, for FL scenarios, the training efficiency can also be enhanced by adaptively tuning the synchronization frequency [99], [100]. Those approaches can be adopted together with our LB-BSP scheme.

7 Discussions

Data Access Frequency. Note that in this paper we focus on cases where each worker can access the *entire* dataset, via either local storage or Network File System (NFS) [101], [102], [103]. In shared production clusters, it has become almost a norm to store data in NFS (the access delay can be made negligible via prefetching), which largely facilitates data management and job migration [10], [11], [12], [30]. Nonetheless, there may still exist some cases that each worker can only access a local partition of the training dataset. Under LB-BSP where faster workers iterate with larger batches, this means that samples on faster workers would be accessed more frequently. Such a problem of *uneven sample access frequency* may lead to inaccurate training results, especially when the dataset is not well shuffled before being partitioned and there are huge gaps on worker processing capability. To address uneven sample access frequency with transparency to the upper level training process, we suggest an iterative SSP-style data scheduling scheme: once the traversal times of one partition exceed another over a given *threshold*, we launch a background process to migrate certain amount of samples from the slower worker to the faster one. We have prototyped[13] this method atop PyTorch, and verified that it could make ideal accuracy even under

13. We develop a Python module called `DataPartitionManager` to periodically collect each partition's traversal status and launch peer-to-peer data transmission when appropriate, where all the communications are in Thrift RPC calls. Once a data shifting process finishes, we reset the input stream (e.g., `DataLoader` in PyTorch) and partition-traversal statistics on all the workers. In our verification, we train the ResNet-32 model with 5 workers; each worker hosts two classes of the CIFAR-10 dataset and their batch sizes are 64, 96, 128, 160 and 192, respectively. With the `DataPartitionManager` and the gap of traversal times bounded by 2, we obtain the same test accuracy (0.92) as training with *i.i.d* dataset.

non-i.i.d. data distribution. Such a kind of data migration method is indeed light-weight in GPU clusters, because in GPU clusters the worker speed is relatively stable and data migration is done-once-and-working-forever, with the overhead amortized.

ASIC Hardware. While in this work LB-BSP is designed mainly for CPU and GPU clusters, it can also be adopted for Application Specific Integrated Circuit (ASIC) hardware like TPU or FPGA. Existing measurements in the literature [104], [105] have shown that TPU and FPGA also exhibit a *non-linear*, *monotonically-increasing* relationship between the batch computation time and batch size, identical with that of a GPU (as shown in Fig. 4). Therefore, our LB-BSP algorithm for GPU clusters, i.e., Algorithm 2, should be able to work well in heterogeneous clusters with such ASIC hardware.

Semi-Dynamic Load Balancing With ASP or SSP. Although LB-BSP is built atop BSP, its basic philosophy, semi-dynamic load balancing, can also be integrated into ASP and SSP to avoid stale parameters. Nonetheless, under ASP or SSP each worker would proceed without per-iteration barriers, and it is thus hard to *simultaneously* and *coordinately* adjust each worker's batch size, leading to a larger overhead and worse load-balancing effect.

End-of-Iteration Network Contention. Since LB-BSP can equalize ML workers' batch processing time, this would however intensify the network contention at the iteration boundaries. In scenarios where the network bandwidth is a severe bottleneck, it is highly suggested that LB-BSP is adopted along with certain communication optimization techniques as elaborated in Section 6.

8 Concluding Remarks

In this work, we find that machine learning workloads should be load-balanced in a semi-dynamic manner, and have proposed LB-BSP to achieve efficient distributed learning with non-dedicated resources. LB-BSP works by speculatively apportioning the load on each worker based on its temporal processing capability. We have designed LB-BSP respectively for CPU clusters, GPU clusters and FL setups. Our testbed experiments show clear evidence that LB-BSP can effectively eliminate stragglers in non-dedicated clusters, speeding up model convergence by over 50 percent.

Acknowledgments

This work was supported by RGC GRF under Grants 16206417, 16207818, and 26213818. The work of Qizhen Weng was supported by the Hong Kong PhD Fellowship Scheme.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Conf. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.
[2] C. Farabet, C. Couprie, L. Najman, and Y. LeCun, "Learning hierarchical features for scene labeling," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 35, no. 8, pp. 1915–1929, Aug. 2013.
[3] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, "Natural language processing (almost) from scratch," *J. Mach. Learn. Res.*, vol. 12, pp. 2493–2537, 2011.
[4] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proc. Conf. Neural Inf. Process. Syst.*, 2014, pp. 3104–3112.
[5] J. Dean *et al.*, "Large scale distributed deep networks," in *Proc. Conf. Neural Inf. Process. Syst.*, 2012, pp. 1223–1231.
[6] M. Li *et al.*, "Scaling distributed machine learning with the parameter server," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2014, pp. 583–598.
[7] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project Adam: Building an efficient and scalable deep learning training system," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2014, pp. 571–582.
[8] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2016, pp. 265–283.
[9] EC2 Spot Instances. 2020. [Online]. Available: https://aws.amazon.com/ec2/spot/
[10] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, "Analysis of large-scale multi-tenant GPU clusters for DNN training workloads," in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 947–960.
[11] W. Xiao, *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2018, pp. 595–610.
[12] K. Hazelwood *et al.*, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2018, pp. 620–629.
[13] Q. Ho *et al.*, "More effective distributed ML via a stale synchronous parallel parameter server," in *Proc. Conf. Neural Inf. Process. Syst.*, 2013, pp. 1223–1231.
[14] H. Cui *et al.*, "Exploiting bounded staleness to speed up big data analytics," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, 2014, pp. 37–48.
[15] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, 1999.
[16] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing," in *IEEE/ACM Proc. Conf. High Perform. Comput. Netw., Storage Anal.*, 2009, pp. 1–11.
[17] J. Dean and S. Ghemawat, "Mapreduce: A flexible data processing tool," *Commun. ACM*, vol. 53, no. 1, pp. 72–77, 2010.
[18] T. Kokilavani *et al.*, "Load balanced min-min algorithm for static meta-task scheduling in grid computing," *Int. J. Comput. Appl.*, vol. 20, no. 2, pp. 43–49, 2011.
[19] P. Samal and P. Mishra, "Analysis of variants in round robin algorithms for load balancing in cloud computing," *Int. J. Comput. Sci. Inf. Technol.*, vol. 4, no. 3, pp. 416–419, 2013.
[20] X. Tang and S. T. Chanson, "Optimizing static job scheduling in a network of heterogeneous computers," in *Proc. IEEE Int. Conf. Parallel Process.*, 2000, pp. 373–382.
[21] J. Yang and Q. He, "Scheduling parallel computations by work stealing: A survey," *Int. J. Parallel Program.*, vol. 46, no. 2, pp. 173–197, 2018.
[22] B. Acun and L. V. Kale, "Mitigating processor variation through dynamic load balancing," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2016, pp. 1073–1076.
[23] A. Harlap *et al.*, "Addressing the straggler problem for iterative convergent parallel ML," in *Proc. 7th ACM Symp. Cloud Comput.*, 2016, pp. 98–111.
[24] U. A. Acar, A. Charguéraud, and M. Rainey, "Scheduling parallel programs by work stealing with private deques," in *ACM SIGPLAN Notices*, vol. 48, no. 8, pp 219–228, 2013.
[25] T. Chen *et al.*, "MXNET: A flexible and efficient machine learning library for heterogeneous distributed systems," 2015, *arXiv:1512.01274*.
[26] PyTorch. 2020. [Online]. Available: https://pytorch.org/
[27] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *in Proc. 7th ACM Symp. Cloud Comput.*, 2012, pp. 1–13.
[28] E. Diaconescu, "The use of NARX neural networks to predict chaotic time series," *Wseas Trans. Comput. Res.*, vol. 3, no. 3, pp. 182–191, 2008.
[29] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: An efficient dynamic resource scheduler for deep learning clusters," in *Proc. Proc. 13th EuroSys Conf.*, 2018, pp. 1–14.
[30] J. Gu *et al.*, "Tiresias: A GPU cluster manager for distributed deep learning," in *Proc. USENIX Conf. Netw. Syst. Des. Implementation*, 2019, pp. 485–500.
[31] C. Chen, Q. Weng, W. Wang, B. Li, and B. Li, "Semi-dynamic load balancing: Efficient distributed learning in non-dedicated environments," in *Proc. 11th ACM Symp. Cloud Comput.*, 2020, pp. 431–446.

[32] J. Konečný, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, "Federated learning: Strategies for improving communication efficiency," 2016, *arXiv:1610.05492*.

[33] C. Zhang, S. Li, J. Xia, W. Wang, F. Yan, and Y. Liu, "BatchCrypt: Efficient homomorphic encryption for cross-silo federated learning," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, 2020, pp. 493–506.

[34] Q. Yang, Y. Liu, T. Chen, and Y. Tong, "Federated machine learning: Concept and applications," *ACM Trans. Intell. Syst. Technol.*, vol. 10, no. 2, pp. 1–19, 2019..

[35] M. Li, T. Zhang, Y. Chen, and A. J. Smola, "Efficient mini-batch training for stochastic optimization," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2014, pp. 661–670.

[36] H. Zhang *et al.*, "Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, 2017, pp. 181–193.

[37] P. Goyal *et al.*, "Accurate, large minibatch SGD: Training imagenet in 1 hour," 2017, *arXiv:1706.02677*.

[38] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing, "Geeps: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server," in *Proc. ACM Eur. Conf. Comput. Syst.*, 2016, pp. 1–16.

[39] X. Jia *et al.*, "Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes," 2018, *arXiv:1807.11205*.

[40] T. Li, J. Zhong, J. Liu, W. Wu, and C. Zhang, "Ease. ml: Towards multi-tenant resource sharing for machine learning workloads," *Proc. VLDB Endowment*, vol. 11, no. 5, pp. 607–620, 2018.

[41] J. H. Park *et al.*, "Hetpipe: Enabling large DNN training on (whimpy) heterogeneous GPU clusters through integration of pipelined model parallelism and data parallelism," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, 2020, pp. 307–321.

[42] S. Chaudhary, R. Ramjee, M. Sivathanu, N. Kwatra, and S. Viswanatha, "Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning," in *Proc. 15th Eur. Conf. Comput. Syst.*, 2020, pp. 1–16.

[43] K. Mahajan *et al.*, "Themis: Fair and efficient GPU cluster scheduling," in *Proc. USENIX Conf. Netw. Syst. Des. Implementation*, 2020, pp. 289–304.

[44] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia, "Heterogeneity-aware cluster scheduling policies for deep learning workloads," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2020, pp. 481–498.

[45] K. Bonawitz *et al.*, "Towards federated learning at scale: System design," 2020, *arXiv:1902.01046*.

[46] J. Langford, A. J. Smola, and M. Zinkevich, "Slow learners are fast," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2009, pp. 2331–2339.

[47] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones," in *Proc. USENIX Conf. Netw. Syst. Des. Implementation*, 2013, pp. 185–198.

[48] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2008, pp. 29–42.

[49] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. USENIX Conf. Netw. Syst. Des. Implementation*, 2012, p. 2.

[50] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous SGD," 2016, *arXiv:1604.00981*.

[51] Y. Jia *et al.* "Caffe: Convolutional architecture for fast feature embedding," 2014, *arXiv:1408.5093*.

[52] K. Al Nuaimi, N. Mohamed, M. Al Nuaimi, and J. Al-Jaroodi, "A survey of load balancing in cloud computing: Challenges and algorithms," in *Proc. Symp. Netw. Cloud Comput. Appl.*, 2012, pp. 137–142.

[53] A. N. Tantawi and D. Towsley, "Optimal static load balancing in distributed computer systems," *J. ACM*, vol. 32, no. 2, pp. 445–465, 1985.

[54] J. Jiang, B. Cui, C. Zhang, and L. Yu, "Heterogeneity-aware distributed parameter servers," in *Proc. ACM Int. Conf. Manage. Data*, 2017, pp. 463–478.

[55] H. Zhang, L. Stafman, A. Or, and M. J. Freedman, "SLAQ: Quality-driven scheduling for distributed machine learning," in *Proc. Symp. Cloud Comput.*, 2017, pp. 390–404.

[56] T. Le, X. S. Sun, M. Chowdhury, and Z. Liu, "Allox: Compute allocation in hybrid clusters," *Proc. 15th Eur. Conf. Comput. Syst.*, 2020, pp. 1–16.

[57] Stress-ng: a tool to load and stress a computer system. 2019. [Online]. Available: http://manpages.ubuntu.com/manpages/artful/man1/stress-ng.1.html

[58] V. Ş. Ediger and S. Akar, "Arima forecasting of primary energy demand by fuel in turkey," *Energy Policy*, vol. 35, no. 3, pp. 1701–1708, 2007.

[59] J. T. Connor, R. D. Martin, and L. E. Atlas, "Recurrent neural networks and robust time series prediction," *IEEE Trans. Neural Netw.*, vol. 5, no. 2, pp. 240–254, Mar. 1994.

[60] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[61] A. M. Rather, A. Agarwal, and V. Sastry, "Recurrent neural network and a hybrid model for prediction of stock returns," *Expert Syst. Appl.*, vol. 42, no. 6, pp. 3234–3241, 2015.

[62] N. G. Polson and V. O. Sokolov, "Deep learning for short-term traffic flow prediction," *Transp. Res. Part C: Emerg. Technol.*, vol. 79, pp. 1–17, 2017.

[63] Y. Gao and M. J. Er, "Narmax-model-based time series modeling and prediction: Feedforward and recurrent fuzzy neural network approaches," in *WSEAS CSECS*, pp. 331–350, 2003.

[64] D. Argyropoulos, D. S. Paraforos, R. Alex, H. W. Griepentrog, and J. Müller, "Narx neural network modelling of mushroom dynamic vapour sorption kinetics," *IFAC-PapersOnLine*, vol. 49, no. 16, pp. 305–310, 2016.

[65] E. Cadenas, W. Rivera, R. Campos-Amezcua, and R. Cadenas, "Wind speed forecasting using the NARX model, case: La mata, oaxaca, méxico," *Neural Comput. Appl.*, vol. 27, no. 8, pp. 2417–2428, 2016.

[66] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," Univ, Toronto, Toronto, ON, Canada, Tech. Rep., 2009.

[67] D. Lustig and M. Martonosi, "Reducing GPU offload latency via fine-grained CPU-GPU synchronization," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2013, pp. 354–365.

[68] A. Magni, C. Dubach, and M. O'Boyle, "Exploiting GPU hardware saturation for fast compiler optimization," in *Proc. ACM Proc. Workshop General Purpose Process Using GPUs*, 2014, pp. 99–106.

[69] A. Li, "GPU performance modeling and optimization," Ph.D. dissertation, Dept, Electrical Eng., Technische Universiteit Eindhoven, Eindhoven, The Netherlands, 2016.

[70] J. Gu, H. Liu, Y. Zhou, and X. Wang, "Deepprof: Performance analysis for deep learning applications via mining GPU execution patterns," 2017, *arXiv:1707.03750*.

[71] J. Konečný, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, "Federated Learning: Strategies for Improving Communication Efficiency," 2017, *arXiv:1610.05492*.

[72] K. Cheng, T. Fan, Y. Jin, Y. Liu, T. Chen, and Q. Yang, "Secureboost: A lossless federated learning framework," 2019, *arXiv:1901.08755*.

[73] Python Psutil. 2020. [Online]. Available: https://psutil.readthedocs.io/en/latest/

[74] Apache Thrift. 2020. [Online]. Available: https://thrift.apache.org/

[75] Keras. 2020. [Online]. Available: https://keras.io/

[76] Y. Yang, D.-C. Zhan, Y. Fan, Y. Jiang, and Z.-H. Zhou, "Deep learning for fixed model reuse," in *Proc. AAAI Conf. Artificial Intell.*, 2017, pp. 2831–2837.

[77] Y. Yao, L. Rosasco, and A. Caponnetto, "On early stopping in gradient descent learning," *Constructive Approx.*, vol. 26, no. 2, pp. 289–315, 2007.

[78] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.

[79] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 2818–2826.

[80] O. Russakovsky *et al.*, "ImageNet Large Scale Visual Recognition Challenge," *Int. J. Comput. Vis.*, vol. 115, no. 3, pp. 211–252, 2015.

[81] Wonder Shaper. 2020. [Online]. Available: https://github.com/magnific0/wondershaper

[82] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms," *Proc. Symp. Operating Syst. Princ.*, 2017, pp. 153–167.

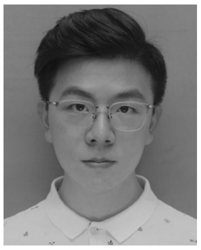
[83] J. Ma, L. K. Saul, S. Savage, and G. M. Voelker, "Identifying suspicious URLs: An application of large-scale online learning," in *Proc. Annu. Int. Conf. Mach. Learn.*, 2009, pp. 681–688.

[84] T. Ozaki, "On the order determination of ARIMA models," *Appli. Stat.*, pp. 290–301, 1977.

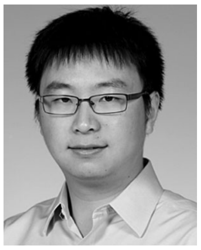
[85] Y. LeCun *et al.*, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[86] P. Warden, "Speech commands: A dataset for limited-vocabulary speech recognition," 2018, *arXiv:1804.03209*.
[87] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proc. ACM Symp. Operating Syst. Princ.*, 2013, pp. 423–438.
[88] A. Toshniwal, *et al.*, "Storm@ twitter," in *ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 147–156.
[89] T. Das, Y. Zhong, I. Stoica, and S. Shenker, "Adaptive stream processing using dynamic batch sizing," in *Proc. Symp. Cloud Comput.*, 2014, pp. 1–13.
[90] Q. Zhang, Y. Song, R. R. Routray, and W. Shi, "Adaptive block and batch sizing for batched stream processing system," in *Proc. IEEE Int. Conf. Autonomic Comput.*, 2016, pp. 35–44.
[91] A. Devarakonda, M. Naumov, and M. Garland, "AdaBatch: Adaptive batch sizes for training deep neural networks," 2017, *arXiv:1712.02029*.
[92] S. De, A. Yadav, D. Jacobs, and T. Goldstein, "Automated inference with adaptive batches," in *Proc. Artif. Intell. Stat.*, 2017, pp. 1504–1513.
[93] S. Shi and X. Chu, "MG-WFBP: Efficient data communication for distributed synchronous SGD algorithms," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, 2019, pp. 172–180.
[94] C. Chen, W. Wang, and B. Li, "Round-Robin Synchronization: Mitigating Communication Bottlenecks in Parameter Servers," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, 2019, pp. 532–540.
[95] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "1-bit stochastic gradient descent and its application to data-parallel distributed training of speech DNNs," in *Proc. 15th Annu. Confe. Int. Speech Commun. Assoc.*, 2014, pp. 1058–1062.
[96] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "QSGD: Communication-efficient SGD via gradient quantization and encoding," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 1709–1720.
[97] N. Strom, "Scalable distributed DNN training using commodity GPU cloud computing," in *Proc. 16th Annu. Conf. Int. Speech Commun. Assoc.*, 2015, pp. 1488–1492.
[98] K. Hsieh *et al.*, "Gaia: Geo-distributed machine learning approaching LAN speeds," in *Proc. 14th USENIX Symp. Netw. Syst. Des. Implementation*, 2017, pp. 629–647.
[99] J. Wang and G. Joshi, "Adaptive communication strategies to achieve the best error-runtime trade-off in local-update SGD," 2019, *arXiv:1810.08313*.
[100] S. Wang *et al.*, "Adaptive federated learning in resource constrained edge computing systems," *J. Sel. Areas Commun.*, vol. 37, no. 6, pp. 1205–1221, 2019.
[101] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proc. Symp. Mass Storage Syst. Technol.*, 2010, pp. 1–10.
[102] GlusterFS 2020. [Online]. Available: https://docs.gluster.org/en/latest/
[103] F. B. Schmuck and R. L. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proc. USENIX Conf. File Storage Technol.*, 2002, vol. 2, no. 19.
[104] Y. Kochura *et al.*, "Batch size influence on performance of graphic and tensor processing units during training and inference phases," 2018, *arXiv:1812.11731*.
[105] T. Posewsky and D. Ziener, "Throughput optimizations for FPGA-based deep neural network inference," *Microprocessors Microsyst.*, vol. 60, pp. 151–161, 2018.

Chen Chen (Member, IEEE) received the BEng degree from Tsinghua University in 2014 and the PhD degree from the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, in 2018. He is currently a researcher with Theory Lab, Huawei HK. His current research interests include distributed deep learning, big data systems, and networking.



Qizhen Weng (Student Member, IEEE) received the BEng (Hons.) degree from the School of Information Security Engineering, Shanghai Jiao Tong University, in 2017. He is currently working toward the PhD degree with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology. His research interests include cloud computing and big data systems, with a special focus on cluster management with machine learning techniques. He was the recipient of the Hong Kong PhD Fellowship Award.



Wei Wang (Member, IEEE) received the BEng (Hons.) and MEng degrees from the Department of Electrical and Computer Engineering, Shanghai Jiao Tong University, and the PhD degree from the Department of Electrical and Computer Engineering, University of Toronto, in 2015. He is currently an assistant professor with the Department of Computer Science and Engineering, the Hong Kong University of Science and Technology (HKUST) and affiliated with HKUST Big Data Institute. His research interests include distributed systems, with special emphasis on big data and machine learning systems, cloud computing, and computer networks in general. He was the recipient of Best Paper Runner-up Award at USENIX ICAC 2013. He was recognized as the distinguished TPC member of IEEE INFOCOM 2018, 2019, and 2020.

Baochun Li (Fellow, IEEE) received the BEngr degree from the Department of Computer Science and Technology, Tsinghua University, China, in 1995 and the MS and PhD degrees from the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, in 1997 and 2000, respectively. Since 2000, he has been with the Department of Electrical and Computer Engineering, the University of Toronto, where he is currently a professor. Since August 2005, he has been with the Bell Canada Endowed Chair in computer engineering. His research interests include cloud computing, distributed systems, datacenter networking, and wireless systems. He was the recipient of IEEE Communications Society Leonard G. Abraham Award in the Field of Communications Systems in 2000, Multimedia Communications Best Paper Award from the IEEE Communications Society in 2009, and University of Toronto McLean Award. He is a member of ACM.

Bo Li (Fellow, IEEE) received the BEng (summa cum laude) degree in computer science from Tsinghua University, Beijing and the PhD degree in electrical and computer engineering from the University of Massachusetts at Amherst. He is currently a chair professor with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology. Between 2010 and 2016, he was a Cheung Kong visiting chair professor with Shanghai Jiao Tong University and the chief technical advisor for ChinaCache Corporation, a leading CDN provider, and an adjunct researcher with the Microsoft Research Asia from 1999 to 2006 and Microsoft Advanced Technology Center from 2007 to 2008. He made pioneering contributions in multimedia communications and the Internet video broadcast, in particular Coolstreaming system, which was credited as first large-scale Peer-to-Peer live video streaming system in the world. It attracted significant attention from both industry and academia. He has been an editor or a guest editor of more than two dozen of the IEEE and ACM journals and magazines. He was the Co-TPC chair of the IEEE INFOCOM 2004. He was the recipient of Test-of-Time Best Paper Award from IEEE INFOCOM 2015 for his contributions.