Semi-Dynamic Load Balancing: Efficient Distributed Learning in Non-Dedicated Environments

Chen Chen
HKUST
cchenam@connect.ust.hk

Qizhen Weng
HKUST
qwengaa@cse.ust.hk

Wei Wang
HKUST
weiwa@cse.ust.hk

Baochun Li
University of Toronto
bli@ece.toronto.edu

Bo Li
HKUST
bli@cse.ust.hk

ABSTRACT
Machine learning (ML) models are increasingly trained in clusters with non-dedicated workers possessing heterogeneous resources. In such scenarios, model training efficiency can 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 paper, 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. We develop LB-BSP based on such an insight, which is 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 custom-designed the batch sizing algorithm respectively for CPU and GPU clusters based on their own 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%.

CCS CONCEPTS
• Computing methodologies → Distributed computing methodologies; Machine learning.

KEYWORDS
Distributed Learning, Load Balancing, Synchronization

ACM Reference Format:

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 [23, 36, 53, 76]. Owing to their compute-intensive nature, ML models are often trained in a distributed manner [11, 22, 30, 57]: 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 [2], or with time-varying resources from shared production clusters [42, 46, 81]. 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 [26, 30, 44] 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 (§2.2.3). Yet, different from those traditional workloads such as HPC processing [17, 34] or MapReduce [31], 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 [51, 73, 78] are not aware of runtime resource variations, and dynamic approaches [13, 41, 82], which are mainly based on work stealing [12, 17, 34], 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 samples are processed one by one, which is indeed incompatible with the style of tensor-based processing in modern ML frameworks [9, 11, 21].

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 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 (§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 [42, 71], and thus shall be predicted prior to each iteration. For this purpose, we employ a special kind of recurrent neural network called NARX [33], which can make accurate performance predictions by taking into account the driving parameters such as CPU and memory.

In multi-tenant GPU clusters [39, 46, 67, 81], 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 (§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.

We have implemented LB-BSP with a Python module that can be easily integrated into existing ML frameworks like TensorFlow [11], PyTorch [9] and MXNet [21]. 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%; and in a 32-node shared CPU cluster, it attains an improvement of up to 38.7% over state-of-the-art 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 $\omega^*$ that can minimize the loss function $L(\omega)$ over the labeled training dataset $S$, i.e.,

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

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 [30, 58], or simply SGD\(^1\). Its basic idea is to iteratively refine model parameters $\omega$ with the gradients $g$ calculated from the sample batches, i.e.,

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

Here, $k$ is the iteration number, $B^k$ is a batch randomly sampled from the training dataset $S$, and $\eta$ is the learning rate. Such training iterations would repeat until the model parameters $\omega$ 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

\(^1\)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 (§3.4).
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 to-
wards model convergence. Therefore, efficient model training 
requires both high hardware efficiency and high statistical 
efficiency.

**Distributed Model Training.** To accelerate the model train-
ing process, it has been increasingly common to train ML 
models in a distributed manner, with a pool of parallel work-
ers. Those workers typically collaborate in a synchronous 
mode (i.e., BSP) [27, 38, 89], under the Parameter Server 
(PS) [27, 44, 57] or All-Reduce [38] architecture.

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

1. **Dedicated clusters** implies that the clusters are homo-
ogeneous, 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 [38] 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 [38, 47].

2. **Non-dedicated clusters** refer to the clusters that are 
composed of heterogeneous hardware or shared by multiple 
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 [59, 66] or from spot markets such 
as Amazon EC2 [2, 4, 49] or FloydHub [5]. Meanwhile, large 
companies [42, 46] may host multiple ML jobs in their produc-
tion clusters with mixed hardware types; for fairness, users 
may be allocated heterogeneous hardware [19, 63, 64], and 
their resource allocation may also be dynamically adjusted 
by cluster schedulers for resource packing purpose [39, 81].

While model training in non-dedicated clusters is increas-
ingly 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 tempo-
rary 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 the hetero-
genrety of worker resource quality or quantity, and are often 
severe and long-lasting. They occur only in non-dedicated 
clusters.

Compared with non-deterministic stragglers, determinis-
tic 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 con-
straint compromised have been proposed: ASynchronous 
Parallel (ASP), and Stale Synchronous Parallel (SSP).

**ASP.** In ASP [30], 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 [44, 54].

**SSP.** SSP [26, 44] 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 stragglers 
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** [15, 87] is widely adopted to miti-
gate stragglers in traditional data analytics frameworks like 
MapReduce [31] and Spark [85]. 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. [20] pro-
posed to train deep neural models with extra backup workers 
and to use gradients from those workers finishing the earli-
est. 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 [11, 21, 48] 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 [14] 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 [51, 73, 78, 79] set a constant load for each worker—as the execution commences—under a given scheme like Round-Robin [73] or with some static knowledge of the worker status [78, 79]. 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 LoadBalancing.** Dynamic load balancing strategies use work-stealing or work-shedding to redistribute load from heavily-loaded workers to lightly-loaded ones at runtime [12, 13, 17, 34, 82]. They are mostly developed for traditional task-based parallel programming models in multi-core or HPC systems. Recently, FlexRR [41] 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 ($3.3.1$). 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 NDArry 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 light-weight 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 Fig. 1 and Fig. 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.
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 elaborate our solutions for CPU and GPU clusters, respectively.

3.1 Problem Formulation

In each model training iteration, given the sample batch, a worker first calculates a local gradient and then remotely elaborates with the problem formulation of LB-BSP, and then elaborate our solutions for CPU and GPU clusters, respectively.

Table 1: Summary of important notations.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t$</td>
<td>batch processing time</td>
</tr>
<tr>
<td>$t^p$</td>
<td>computation time</td>
</tr>
<tr>
<td>$t^m$</td>
<td>communication time</td>
</tr>
<tr>
<td>$\Gamma(\cdot)$</td>
<td>function between $t^p$ and $x$</td>
</tr>
<tr>
<td>$x_i$</td>
<td>batch size on worker-$i$</td>
</tr>
<tr>
<td>$\bar{x}$</td>
<td>initial batch size</td>
</tr>
<tr>
<td>$X$</td>
<td>net $(n$ is worker number$)$</td>
</tr>
<tr>
<td>$\nu$</td>
<td>sample processing speed</td>
</tr>
</tbody>
</table>

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 $\bar{x}$, we want to find the worker batch sizes $\bar{x} = (x_1, x_2, ..., x_n)$ that can minimize the longest batch processing time among all workers, i.e.,

$$
\min_{\bar{x}=(x_1, x_2, ..., x_n)} \max_{i=1,2,...,n} t_i,
\text{ s.t. } t_i = t_i^p + t_i^m, \quad i = 1, \ldots, n; \quad t_i^p = \Gamma_i(x_i), \quad i = 1, \ldots, n; \quad \sum_{i=1}^n x_i = X = nx.
$$

(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 shared 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 $\bar{x}$ for

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

CPU clusters (§3.2), and a numerical method that iteratively approaches the optimal $\bar{x}$ for GPU clusters (§3.3).

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 [41, 42, 49, 88].

(2) Non-urgent Model Training. Non-urgent ML tasks may also be trained in CPU clusters opportunistically with leftover resources [42, 55]. 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 [42].

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 [89] 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 §5.1) with 32 workers and one separate PS (c5.2xlarge instances with ≤ 10Gbps bandwidth), $t^p$ takes more than 99% of the batch processing time $t$.

(2) Linear Relationship: $\Gamma'(x) = x/\nu$. 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 $I \approx t^p = f(x) = x/v$.

Given the characteristics above, we solve optimization problem (3) with $x_i = \frac{\sum_{j=0}^{m-i} z_j}{\sum_{j=0}^{m-i} x_j} \cdot X$. 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) [35] makes predictions with statistics like average and deviation. Meanwhile, models based on Recurrent Neural Network (RNN) [24] (like plain RNN and LSTM [45]), with the ability to maintain inner memory, have been applied in forecasting real-world time series like stock price [70] or transport flow [68].

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 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 and make more accurate prediction. To this end, we find that Nonlinear Autoregressive eXogenous (NARX) model [33, 37] 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 ($v$), current and past values of the two driving resources—CPU and memory usage ($c/m$). 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:

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

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 Alg.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., $v$, $c$ and $m$). To avoid high model complexity, as in existing prediction works [16, 18], 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

---

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.
Algorithm 1 Batch Size Updating in CPU clusters

Input: \([x_i^{k-1}], [t_i^{k-1}], [c_i^j], [m_i^j]\) \text{ last batch size, last batch processing time, current cpu & memory usage of all the workers} (i = 1, 2, ..., n)

Require: \(F_i(\cdot), t_i, c_i, m_i, i = 1, 2, ..., n; \)

\(F_i(\cdot), t_i, c_i, m_i, i = 1, 2, ..., n; \) \(\rightarrow\) NARX model for each worker

1: procedure CPU\_UPDATE\_BATCH\_SIZE(k)
2: \(x_i^{k-1} \leftarrow x_i^{k-1} / t_i^{k-1}, i = 1, 2, ..., n.\)
3: \(x_i^{k} = F_i(t_i^{k-1}, ..., t_i^{k-d_{i}}, c_i^{k-1}, ..., c_i^{k-d_{i}}, m_i^{k-1}, ..., m_i^{k-d_{i}}), i = 1, ..., n.\)
4: \(X \leftarrow \sum_{i=1}^{n} x_i^{k-1}.\)
5: \(x_i^{k} = \frac{x_i^{k-1}}{\sum_{i=1}^{n} x_i^{k}}, \quad X, i = 1, 2, ..., n.\)
6: round \(x_i^{k} (i = 1, ..., n)\) to integers with \(\sum_{i=1}^{n} x_i^{k} = X.\)
7: return \(x_i^{k} \)

avoids over-fitting and converges fast. Our later evaluation (§5.4) confirms that such a NARX-based approach can make better predictions than other approaches we surveyed.

3.3 LB-BSP in GPU Clusters

With strong parallel processing capability, GPUs are the workhorse hardware for training deep neural networks [38, 46, 89]. As elaborated in §2.1, neural network models may be trained in non-dedicated GPU clusters: with heterogeneous GPU instances from the spot markets [2, 5], or in shared GPU clusters [39, 46, 67, 81] where workers may inter-communicate at different locality levels and be migrated across machines for resource consolidation. Nonetheless, implementing LB-BSP in non-dedicated GPU clusters renders a challenge different than 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.

1) Non-negligible Communication Time: \(t^p \gg t^m\) \(t \neq t^p\). 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 [89].

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 [60] 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: \(\Gamma(x) = C, \forall x \leq x^*\). 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 \(x^*\)) to saturate all the processing kernels [56, 62]. Reducing batch size under \(x^*\) does not help to reduce the computation time and would cause GPU underutilization.

(4) GPU Memory Limitation: \(x < x^m\). 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 \(x^m\) on the maximum 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.

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) [40, 46], and auxiliary resources (e.g., CPU, memory, network connectivity) provisioned in GPU clusters fluctuate less often than in CPU clusters [39, 46, 67, 81], 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 Alg. 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% available memory would, to avoid OOM error, be excluded from being identified as the leader. Moreover, if with Alg. 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.

**Remarks.** While Alg. 2 is designed for GPU clusters, it also applies to other accelerators like TPU or FPGA. Existing measurements have shown that TPU [50] and FPGA [69] also exhibit a non-linear, monotonically-increasing relationship between \( t^p \) and \( x \), making it feasible to adopt LB-BSP in heterogeneous clusters with those hardware.

### 3.4 Weighted Gradient Aggregation

By tuning the worker batch size with Alg. 1 and Alg. 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-\( i \) (\( i = 1, 2, ..., n \)), then

\[
g = \frac{1}{n} \sum_{i=1}^{n} g_i, \quad \text{where } g_i = \frac{1}{|B_i|} \sum_{s \in B_i} \nabla I(s, \omega).
\]  

(5)

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

\[
g = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{|B_i|} \sum_{s \in B_i} \nabla I(s, \omega) = \sum_{i=1}^{n} \frac{1}{n|B_i|} \nabla I(s, \omega).
\]  

(6)

This implies that, when workers have different batch sizes \((|B_i|)\), the significance of different samples, i.e., \( \frac{1}{|B_i|} \), 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 Alg. 2 in a 16-node heterogeneous GPU cluster (i.e., Cluster-A in §5.1). After traversing the ImageNet dataset for 20 epochs, the training accuracy only reaches 43%, much worse than that under BSP (59%).

**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

\[
g = \frac{1}{\sum_{j=1}^{n} |B_j|} \sum_{i=1}^{n} |B_i| \cdot g_i = \frac{1}{X} \sum_{i=1}^{n} \sum_{s \in B_i} \frac{1}{|B_i|} \nabla I(s, \omega).
\]  

(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 Discussion on Data Access Frequency

In this paper, we focus on cases where each worker can access the entire dataset via local storage or Network File System.
(NFS) [6, 74, 75]. In shared production clusters, it has become almost a norm to store data in NFS (the access delay can be made negligible via pre-fetching), which largely facilitates data management and job migration [39, 42, 46, 81]. 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 in this method atop PyTorch, and verified that it could make ideal accuracy even under 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.

4 IMPLEMENTATION

We implement LB-BSP with BatchSizeManager, a Python module that can be integrated into existing ML frameworks including TensorFlow [11], PyTorch [9] and MXNet [21].

Architecture Overview. The overall workflow of LB-BSP is described in Alg. 3 and Fig. 5. In the beginning of iteration \( k \), each worker pushes its latest execution information (\( \otimes \) batch processing time \( t^{k-1} \), CPU usage \( \hat{c}^k \), memory usage \( m^k \) ) for CPU clusters, or (\( \otimes \) \( r^{k-1} \), \( m^k \) ) 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.

---

**Algorithm 3 LB-BSP Workflow**

**Worker**: \( i=1, 2, ..., n \):

1. \( \text{procedure WorkerIterate}(k) \)
   1. \( \text{UpdateBatchSizeManager} <\text{states}> \)
   2. \( x_i \leftarrow \text{SetBatchSize} <\text{states}> \)
   3. \( \text{WorkerIterate}(k+1) \)

**Parameter Server (PS):**

1. \( \text{parameterServerIterate}(k) \)
   1. \( \text{aggregate gradient} \)
   \( \sum_{i=1}^{n} \frac{1}{x_i} \text{gradient}_i \)
   2. \( \text{update parameters} \)
   \( w^{k+1} \leftarrow w^k - \eta \text{gradient} \)

**BatchSizeManager:**

1. \( \text{procedure UpdateBatchSize} <\text{states}> \)
2. \( \text{Refer to Alg. 1 (CPU cluster) or Alg. 2 (GPU cluster)}. \)

---

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 (§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 couple 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 Dataloader 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` 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 [3], 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 [7], 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 (§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 [83]—with the models trained even for other workers or ML jobs. Besides, we also adopt early stopping [84] 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 (§5.1) and micro-benchmark (§5.2) evaluations in GPU clusters. Then we resort to model training with leftover resources in production CPU clusters (§5.3), with a deep dive analysis of the NARX prediction approach (§5.4). Finally we evaluate the overhead of our LB-BSP solution in §5.5.

5.1 End-to-End Result in GPU Cluster

**Experimental Setup.** As elaborated in §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). On each instance we run a worker process and a collocated PS shard under TensorFlow 1.4.0.

With Cluster-A, we train the CifarNet [52] and ResNet-32 [43] model on CIFAR-10 dataset, and Inception-V3 [77] model on ImageNet dataset [72] (containing 1.28 million of training images of 1000 classes). 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\(^7\) and LB-BSP, and we defer the comparisons with FlexRR and redundant execution to CPU clusters\(^8\). 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 the 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. Fig. 6a shows that LB-BSP remarkably outperforms BSP and SSP, and this is consistent with our analysis in §2.2. Interestingly, LB-BSP is even 15% better than ASP in hardware efficiency when training Inception-V3—we will explain that with micro-benchmark evaluations in §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× and 2.05× the number of BSP to make that accuracy.

Moreover, ASP and SSP fall behind not only in the convergence speed, but also in 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

\(^7\) 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.

\(^8\) 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).
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\(^7\).

**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%. 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% in Fig. 6a) to 28%. Thus, the more heterogeneous the cluster is, the more necessary it is to adopt LB-BSP for load balancing.

5.2 Micro-benchmark in GPU Cluster

To further understand LB-BSP behavior from the micro level, we scale down Cluster-A to contain only 4 GPU 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 (Alg. 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\(^10\).

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 [10] tool).

\(^7\)Inception-V3 is not trained to the ideal accuracy (78.8%) due to our budget limitation. For reference an existing work [20] has confirmed that BSP can yield a higher final accuracy than ASP even in homogeneous clusters.

\(^10\) 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.


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.

5.3 Evaluation in Shared CPU Cluster

Experimental Setup. As elaborated in §3.2, non-neural-network or not-so-urgent models may be trained in non-dedicated CPU clusters. To evaluate LB-BSP performance in such scenarios, 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 [25, 42, 59, 71]. Cluster-B is built based on a one-hour snapshot of Google Trace [71]. 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 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 [61], and ResNet-32 on CIFAR-10 dataset. The batch size for SVM training is 10% of the whole URL dataset (as in [49]), 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 §5.1. Regarding FlexRR, since it is not open-sourced and its sequential sample processing manner is not supported in current ML frameworks (§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 [41].

Hardware Efficiency. Fig. 9 shows the average iteration time when training SVM and ResNet-32 under different schemes in Cluster-B. For fair comparison, each value in Fig. 9 is normalized by the BSP performance in the corresponding framework (sequential-processing-style PyTorch for FlexRR, and TensorFlow for the others). As in Fig. 9, 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 (§2.2.3). In contrast, LB-BSP can bring a speedup of 62% for SVM and 58% for ResNet-32 over BSP, outperforming the second best (FlexRR) by up to 38.7%. Thus, given that LB-BSP can also make the optimal statistical efficiency (§5.1), it can surely lead to the best convergence efficiency among all the schemes evaluated.

5.4 NARX Performance Deep Dive

In this part, we further evaluate the prediction performance of the NARX model (§3.2.2) we use in Cluster-B.

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. 10. 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

<table>
<thead>
<tr>
<th>Instance Type</th>
<th>CPU, Mem (core, GiB)</th>
<th>Num</th>
</tr>
</thead>
<tbody>
<tr>
<td>m4.2xlarge</td>
<td>(8, 32)</td>
<td>17</td>
</tr>
<tr>
<td>c5.2xlarge</td>
<td>(8, 64)</td>
<td>2</td>
</tr>
<tr>
<td>r5.2xlarge</td>
<td>(8, 64)</td>
<td>2</td>
</tr>
<tr>
<td>m4.4xlarge</td>
<td>(8, 32)</td>
<td>17</td>
</tr>
<tr>
<td>m4.xlarge</td>
<td>(4, 16)</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2: Cluster-B composition.
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 these “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 §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 \( \alpha \) (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 on the model selection techniques [65]. 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% in RMSE and 15% in average iteration time.

<table>
<thead>
<tr>
<th>Method</th>
<th>Configuration</th>
<th>RMSE</th>
<th>Avg. Iter Time (Normalized by BSP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memoryless</td>
<td></td>
<td>11.85</td>
<td>0.58</td>
</tr>
<tr>
<td>EMA</td>
<td>( \alpha = 0.2 )</td>
<td>7.85</td>
<td>0.46</td>
</tr>
<tr>
<td>ARIMA</td>
<td>( (p,d,q) = (2,2,1) )</td>
<td>9.67</td>
<td>0.52</td>
</tr>
<tr>
<td>SimpleRNN</td>
<td>look-back=2</td>
<td>8.34</td>
<td>0.40</td>
</tr>
<tr>
<td>LSTM</td>
<td>look-back=2</td>
<td>9.19</td>
<td>0.57</td>
</tr>
<tr>
<td>NARX</td>
<td>look-back=2</td>
<td>4.78</td>
<td>0.42</td>
</tr>
</tbody>
</table>

Table 3: RMSE and iteration speedup when applying different prediction approaches in LB-BSP.

6 ADDITIONAL RELATED WORK

Besides the related work in §2.2, in this part we discuss some additional related work on batching. Batching is necessary when processing long-lasting streaming inputs or training models with large datasets. For big data streaming systems [80, 86], some works [28, 90] have explored how to adaptively adjust the batching interval when faced with dynamic data rates or operating conditions. For iterative model training, some [29, 32] 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.

7 CONCLUSION

In this work, we propose LB-BSP to load-balance distributed model training workloads in a semi-dynamic manner, by speculatively apportioning the load on the workers according to their temporal processing capability. LB-BSP is tailor-made respectively for both CPU and GPU clusters, and our experiments on Amazon EC2 have shown clear evidence that it can effectively eliminate stragglers in non-dedicated clusters, speeding up model convergence by over 50%.

ACKNOWLEDGMENT

The research was supported in part by RGC GRF grants under the contracts 16206417, 16207818 and 26213818. Qizhen Weng was supported in part by the Hong Kong PhD Fellowship Scheme.
REFERENCES

[37] 2016. Effective Straggler Mitigation: Attack of the Clones.. In USENIX NSDI.
[38] 2009. Scalable work stealing. In IEEE/ACM SC.
[42] 2016. Accelerating parallel programs by work stealing with private deques. In USENIX NSDI.


[77] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In IEEE CVPR.


[84] Yang Yang, De-Chuan Zhan, Ying Fan, Yuan Jiang, and Zhi-Hua Zhou. 2010. The hadoop distributed file system. In IEEE MSST.


[86] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and


