Round-Robin Synchronization: Mitigating Communication Bottlenecks in Parameter Servers

Chen Chen, Wei Wang, Bo Li
Hong Kong University of Science and Technology
{cchenam, weiwa, bli}@cse.ust.hk

***Abstract*—Deep learning is usually performed in GPU clusters where each worker machine iteratively refines the model parameters by communicating the update with the Parameter Server (PS). More often than not, workers communicate in a *synchronous* manner, so as to avoid using out-of-dated parameters and make high-quality refinement in each iteration. However, as all workers synchronize with the PS *simultaneously*, the communication becomes a severe *bottleneck*. To address this problem, in this paper we propose the *Round-Robin Synchronous Parallel* (R^2SP) scheme, which coordinates workers to make updates in an *evenly-gapped*, *round-robin* manner. This way, R^2SP can minimize the network contention at a minimum cost of the refinement quality. We further extend R^2SP to *heterogeneous* clusters by adaptively tuning the *batch size* of each worker based on its processing capability. We have implemented R^2SP as a ready-to-use python library for *status-quo* deep learning frameworks. EC2 deployment in GPU clusters show that R^2SP effectively mitigates the communication bottlenecks, accelerating the training of popular image classification models by up to 25%.**

I. Introduction

The recent years have witnessed the wide success of Deep Learning (DL) [1] in many practical applications, such as image classification [2], [3] and speech recognition [4], [5]. Due to the large volume of data samples, neural networks are often trained in cluster environments under the Parameter Server (PS) architecture [6]–[9], where a number of parallel workers *iteratively* compute model updates using their local *sample batches*, and communicate the updates with the PS to refine the global model parameters.

To speed up the training process, workers in learning clusters are typically equipped with GPU accelerators. Given the relatively stable processing speed of GPU devices, many learning frameworks employ the Bulk Synchronous Parallel (BSP) scheme to coordinate GPU workers, which is shown to yield the best training performance [10]–[12]. Under BSP, workers synchronize with the PS at the end of each iteration and will not proceed until the model parameters are fully updated with all their updates. This ensures that those workers can make high-quality model refinements in each iteration, because they always share the *up-to-date* parameters.

However, in GPU clusters, *communication* is frequently a *bottleneck* [13], [14], which significantly delays the training progress in the presence of a *synchronization barrier*. With GPU accelerators, the training of each iteration quickly completes, but workers still have to wait for a long time to transfer updates to the PS and download the refined parameters from there. In our empirical studies, over 90% of iteration time was used for communication when training a VGG16 model in an EC2 cluster with 10 Gbps links (details in Sec. II). Similar observations have also been made in [13], [14]. It is hence imperative to mitigate the communication bottleneck to accelerate training.

Under the BSP scheme, *network contention* arises as a main source of the communication bottleneck. In clusters with *homogeneous* GPU devices, workers synchronize at *roughly the same time*, contending for the network bandwidth of the PS. Consequently, each worker only transfers at low bandwidth, delaying the entire iteration (i.e., low *hardware-efficiency*). To address this problem, an intuitive approach is to remove the synchronization barrier so that workers can communicate at different time: the reduced chance of network contention results in higher bandwidth share for each worker.

A naive implementation of this approach goes to the Asynchronous Parallel (ASP) scheme, where each worker communicates with the PS *asynchronously* and proceeds to the next iteration without waiting for the others' updates. However, the ASP scheme falls short in two aspects. First, without coordination, network transfers from different workers might *randomly "collide"* with each other, leading to *suboptimal* performance on contention mitigation. Second, without synchronization, the computation often iterates on *stale* model parameters, which may drive the updates away from the optimum and thus require more iterations for the model to converge [15]–[17] (i.e., low *statistical efficiency*).

In this paper, our goal is to (1) *minimize* the network contention in GPU clusters for improved hardware efficiency, while (2) attaining *near-optimal* statistical efficiency by minimizing the extent of parameter staleness. To this end, we propose a simple, yet effective worker-coordinating scheme—*Round-Robin Synchronous Parallel* (R^2SP). Under R^2SP, workers make updates to the PS in a fixed *round-robin* order, and those updates are *evenly staggered* with each other. This approach provides two benefits. First, by evenly staggering workers' network transfers, instantaneous contention for the PS bandwidth can be minimized. Second, by coordinating workers to make updates in a fixed *round-robin* order, R^2SP can bound the parameter staleness by a minimum amount, which, we show both theoretically and empirically, achieves much better statistical efficiency than existing asynchronous schemes.

We further generalize the design of R^2SP to *heterogeneous* clusters consisting of GPUs of various generations or architectures—also a common case in production datacen-

ters [17]–[19]. Owing to hardware heterogeneity, those GPU workers are of diverse computing capabilities. Directly applying R^{2}SP can be *inefficient*: fast workers finish computations quickly, but have to wait long for their turn to make update to the PS. We address this problem with *batch size tuning*. In a nutshell, we assign fast workers with *large sample batches*, so as to keep them busy in the entire iteration. As fast workers now do more useful work without idling under the R^{2}SP scheme, the model convergence speed can be further accelerated.

We have implemented R^{2}SP as a ready-to-use Python library for existing learning frameworks such as TensorFlow [7] and MXNet [8]. EC2 deployment atop 16 GPU-workers (`g3.4xlarge` instances) demonstrates that R^{2}SP can reduce the iteration time by more than 30%. Overall, the training convergence is sped up by 25%. Moreover, evaluations in a heterogeneous EC2 cluster show that, with batch size tuning, R^{2}SP can further speed up model convergence by 40%.

II. Background and Motivation

In this section, we briefly review Deep Learning (DL) and the Parameter Server (PS) architecture, along with the synchronization schemes. We show through empirical studies that communication is a bottleneck for GPU-assisted training, of which network contention is the main cause.

A. Background

Deep Learning. Given a neural network model, the goal of DL is to find the model parameters $\omega^\star$ that minimizes the loss function $L(\omega)$ over the entire training dataset, i.e.,

$$\omega^\star = \arg\min L(\omega) = \arg\min \frac{1}{|D|} \sum\nolimits_{s \in D} f(s, \omega).$$

Here, D is the labeled training dataset, and $f(s, \omega)$ is the loss value when using the parameters ω to predict sample s. Typically, f is defined as the sum of the classification cross-entropy and the regularization penalty.

Mini-batch Stochastic Gradient Descent [20], [21], or simply SGD, is the state-of-the-art algorithm to train model parameters $\omega^\star$. Its basic idea is to *iteratively* refine ω with the worker updates, i.e., for iteration k we have $\omega_{k+1} = \omega_k + u_k$, where

$$u_k = -\eta_k g_k = -\eta_k \frac{1}{|I_k|} \sum\nolimits_{s \in I_k} \nabla f(s, \omega_k).$$

Here, u_k is the update; η_k is the learning rate; g_k is the *gradient* calculated from I_k—a sample batch from the training dataset.

Parameter Server. Many real-world learning problems have very large datasets. In this case, the training of DL models is usually performed in a cluster of machines following the Parameter Server (PS) architecture [6]–[8], [20], [22]. In the PS-based systems, the training datasets are distributed to a number of *worker* machines, and the model parameters are stored on one or multiple servers. In each iteration, a worker first *pulls* the latest model parameters from the PS, then computes the local update (gradient) based on its sample batch, and finally *pushes* the gradient to the PS to refine the parameters.

	ResNet32	AlexNet	VGG16	Inception-v3
Parameter Size	50 MB	240 MB	552 MB	108 MB
Communication-Blocking Time	$0.11s$	$2.89s$	$9.04s$	$1.27s$
Iteration Time	$0.23s$	$3.39s$	$9.74s$	$1.97s$
Communication Overhead	47.8%	85.2%	92.8%	64.5%

TABLE I: Communication dominates the learning time for various DL models in a distributed cluster with 8 GPU-workers.

Synchronization Schemes. In the PS-based systems, how to coordinate workers critically impacts the training performance. Three schemes have been developed in this regard: Bulk Synchronous Parallel (BSP), Asynchronous Parallel (ASP), and Stale Synchronous Parallel (SSP).

BSP imposes a *synchronization barrier*, with which workers cannot proceed to the next iteration before the parameters are fully updated. In contrast, ASP removes this barrier and allows workers to *asynchronously* start the next iteration without waiting for the updates from slowed workers. While asynchrony improves hardware efficiency (i.e., makes *short* iterations), it harms statistical efficiency (i.e., requires *more* iterations for model convergence) as the computation may use out-of-date (stale) parameters. SSP [15], [23] comes as a middle ground between the two schemes. It allows asynchronous training with stale parameters, provided that the *progress gap* between the fastest worker and the slowest is within a bounded amount.

Prior work [15], [23], [24] shows that in CPU clusters, relaxed synchronization (e.g., ASP or SSP) usually results in faster learning convergence. However, this is not the case for GPU-assisted training. As computation is no longer the bottleneck (GPUs are *orders-of-magnitude* faster than CPUs in training), improving statistical efficiency becomes more relevant. In GPU clusters, the harm caused by stale parameters *outweighs* the harm from synchronization waiting [10], [11]. Therefore, for GPU-assisted training, the BSP scheme has been widely deployed in practice [11]–[13].

B. Communication Bottlenecks in Parameter Servers

However, the biggest concern about the BSP scheme in GPU clusters is the potential communication bottleneck [13], [14]. As the training in each iteration gets dramatic speedup (usually in sub-seconds) by GPU accelerators, workers and the PS need to frequently exchange updates/parameters.

We empirically quantify the impact of communications through EC2 deployment with 2 PS nodes[1] and 8 GPU workers interconnected by 10 Gbps links. Each PS node is a `c5.9xlarge` instance (36 vCPUs and 72 GB RAM); each worker node is a `g3.4xlarge` instance (one NVIDIA Tesla M60 GPU and 16 GB GPU memory). We trained four popular image classification models in TensorFlow [7] using the BSP scheme: ResNet32, AlexNet, VGG16, and Inception-v3. We used CIFAR-10 [25] datasets for the first three models and

[1]Communication is still a bottleneck when one PS shard is *co-located* with one worker, because in synchronous mode, all the workers would *concurrently* pull/push the parameters stored in *one* PS shard *at a time*.

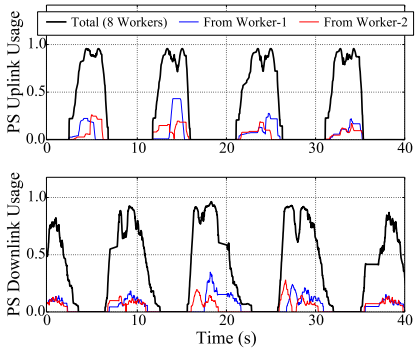


Fig. 1: Total and worker-1/2-originated bandwidth utilization of the PS's downlink and uplink (VGG16).

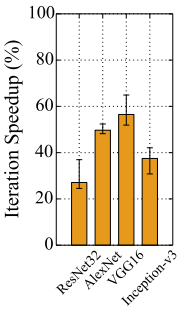


Fig. 2: Iteration speedup if the network contention is eliminated.

ImageNet [26] for the last. The model parameters are equally partitioned between the two PS nodes. We trained each model in 100 iterations and measured the mean iteration time and the *communication-blocking time*. The latter measures how long in an iteration the training is blocked by communications between workers and the PS.

Table I summarizes our measurement results. Across the four models, workers spent most of time communicating with the PS nodes. Notably, in AlexNet and VGG16 models, the *communication overhead*, defined as the communication-blocking time normalized by the iteration time, even exceeds 85% and 90%, respectively.

C. Network Contention under the BSP Scheme

To understand how the communication bottleneck is formed under the BSP scheme and why it is so dominant, we resort to a deep-dive experiment focusing on the network behaviors between workers and the PS. In particular, for each PS node, we measured the bandwidth usage of its *uplink* (outbound to workers) and *downlink* (inbound to PS) during the training process. Fig. 1 depicts our measurement results for VGG16 in a randomly selected interval spanning 40 s. For ease of presentation, we also depict the total amount of traffics contributed by worker-1 and worker-2. The bandwidth usage statistics were collected every 100 ms. We make two observations as follows:

1) *Synchronized communication results in severe network contention, forcing each worker to transfer at low bandwidth.* Under the BSP scheme, workers synchronize at the end of each iteration and compete for the link bandwidth of the PS. As shown in Fig. 1, each worker only receives $1/8$ of the total available bandwidth in both the uplink and downlink, substantially delaying the network transfer.
2) *Synchronization results in uni-directional traffics at a time, wasting bandwidth in full-duplex links.* Network connections in today's datacenters are *full-duplex* links [27], [28], meaning that the uplink and downlink bandwidth are two independent resources that can be utilized *concurrently*. In the presence of the synchronization barrier, however, workers are forced to communicate in one direction at a time—all *pushing* updates before the barrier and *pulling* parameters after it. Therefore, as shown in Fig. 1, more than 50% of time the uplink/downlink is wasted in idle, even though network remains a bottleneck.

In summary, the synchronization barrier imposed by the BSP scheme results in intense network contentions along with low bandwidth utilization. Intuitively, if such contentions can be avoided, we would expect salient training speedup. To illustrate the potential benefits of doing so, we consider an extreme setting where we purposely disabled 7 workers but used only one to perform training, for which there is no network contention. We trained four DL models and measured the iteration time the worker spent. As shown in Fig. 2, the worker is capable of achieving 25-60% speedup over the original setting where it has to contend for bandwidth against the other 7 workers.[2] This promising result motivates us to explore ways to mitigate network contentions via relaxed synchronization.

However, existing schemes with relaxed or no synchronization, notably SSP and ASP, suffer from low statistical efficiency caused by stale parameters [15]. In fact, these schemes result in even slower convergence than that of the BSP scheme for GPU-assisted training [10], [11]. Moreover, neither SSP nor ASP is capable of *minimizing* bandwidth contentions. Without explicit coordinations, workers may have *randomly-overlapped* communications from time to time.

Following the above discussions, we ask: can we have a *contention mitigation* scheme that *improves hardware efficiency* while still *retaining near-optimal statistical efficiency*? We give an affirmative answer to this question in the following sections.

III. Round-Robin Synchronization

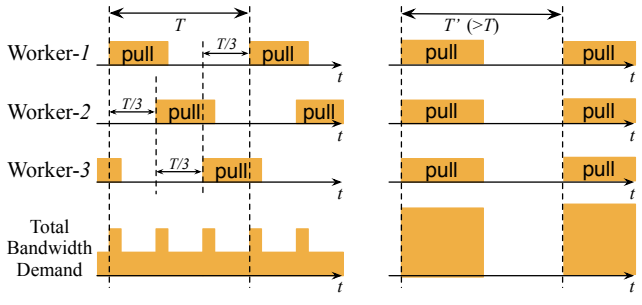
In this section, we present *Round-Robin Synchronous Parallel* (R^2SP) scheme for easing the communication bottleneck at a minimum cost of statistical efficiency. We first limit our discussion to *homogeneous* GPU workers, and defer a more general solution for *heterogeneous* clusters to Sec. IV.

A. Overview

Ideally, our solution should achieve two objectives. First, it should minimize the network contentions to accelerate DL iterations. Second, as this can only be achieved via relaxed synchronization, we shall minimize the negative impact of such relaxation on the statistical efficiency, by restricting the parameter staleness caused.

The R^2SP scheme achieves these two objectives through two control mechanisms that respectively control the *temporal gap* between consecutive updates and the *update order*. Specifically, to minimize network contention, the R^2SP scheme enforces a temporal gap between consecutive updates so as to *evenly stagger* worker updates throughout the training process. Meanwhile, to minimize the parameter staleness, R^2SP coordinates workers to make gradient updates in a *fixed round-robin order*. We next elaborate the two control mechanisms in detail.

[2]The price paid is the reduced image processing throughput as only one worker is used for training. The purpose of this experiment is to illustrate the potential benefits of eliminating network contention.



(a) R^{2}SP (b) BSP

Fig. 3: Under R^{2}SP, the three workers make updates and start to pull parameters at relative time 0, $\frac{T}{3}$ and $\frac{2T}{3}$, respectively. With such an inter-update gap, the network contention (described as *total bandwidth demand*) can be remarkably alleviated compared with that under BSP.

B. Controlling Inter-update Gap for Minimum Contention

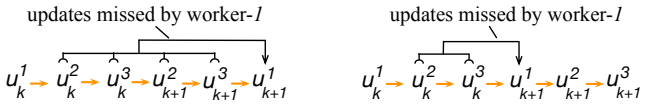
To minimize network contentions, we shall minimize the number of contending workers that communicate with the PS simultaneously. This can be achieved by evenly staggering the worker-PS communications throughout the training process. To do so, the R^{2}SP scheme enforces a temporal *gap* between two consecutive worker updates made to the PS. More precisely, let N be the number of workers and T the timespan of a training iteration. In a homogeneous cluster with a uniform batch size, each GPU worker has the same timespan T as it iterates over the same number of data samples. The PS nodes hence expect N updates received in a period of T time, and the gap between two consecutive updates is set to T/N.

We refer to Fig. 3a as an illustrative example. It depicts how three workers iteratively update their gradients and pull the latest model parameters from the PS under the R^{2}SP scheme. The three workers are scheduled to synchronize with the PS individually, each separated by an inter-update gap of $\frac{T}{3}$. This coordination results in no more than two contending workers throughout the training iterations. In comparison, with the BSP scheme shown in Fig. 3b, all three workers contend for bandwidth at the same time, leading to prolonged communication time due to severe network congestion.

C. Controlling Worker Update Order for Minimum Staleness

While enforcing the above inter-update gap can mitigate the communication bottleneck, the resultant asynchrony poisons statistical efficiency. Next, we further show that, by controlling the worker update *order*, R^{2}SP can minimize the negative effect of such asynchrony on statistical efficiency.

Asynchrony impairs the statistical efficiency due to the problem of stale parameters. Under an asynchronous scheme, a worker calculates its local update (gradient) without seeing the latest updates from the others. Hence, that update is actually based on *stale*, out-of-date parameters, which inevitably diverges the model refinement away from the optimum [15]–[17]. In particular, the *more* updates a worker misses, the *more poisonous* its update is, and in general the *lower* the statistical



(a) SSP (Bound on Progress Gap: 1) (b) R^{2}SP

Fig. 4: When workers make updates asynchronously, each worker would miss some recent updates from others. Under SSP, given 3 workers and the bound on progress gap being 1, the *worst-case staleness* is 4; under R^{2}SP, by enforcing workers to updated in a fixed round-robin manner, that *worst-case staleness* is reduced to 2.

efficiency of the training process [15], [17]. Therefore, to preserve good statistical efficiency, we shall try to minimize the *worst-case staleness*, which is defined as the *maximum* number of updates possibly missed by any worker.

An intuitive solution is to impose a tight *bound* on the *progress gap* (measured by the number of iterations) between the fastest worker and the slowest one, similar to SSP [15]. Fig. 4a depicts an example where the progress gap is bounded by 1 iteration, the minimum one can expect for asynchronous schemes. In the figure, we denote by u_k^i the k^{th} update worker-i makes at the end of the k^{th} iteration. Owing to the bound, worker-1 (i.e., the fastest worker) after pushing its k^{th} update, can proceed to the $(k+1)^{\text{th}}$ iteration only when the other two workers have already entered the k^{th} iteration.

However, imposing such a tight, minimum bound on the progress gap is still *suboptimal* for minimizing staleness, because it suffers from the problem of *out-of-order updates*. As a possible update sequence, in Fig. 4a, while worker-1 is the first to push the k^{th} update among the three workers, its next update u_{k+1}^1 comes as the last among the three. Such a *disorder* does not violate the bound on progress gap, but degrades the worst-case staleness to 4 (in Fig. 4a, worker-1 has missed 4 updates when reporting u_{k+1}^1).

To address this problem, in R^{2}SP we require that workers make updates in a *fixed round-robin* order. As shown in Fig. 4b, in any iteration k the three workers push updates in the order of u_k^1, u_k^2, u_k^3. This results in the minimum worst-case staleness: each worker misses *only two* updates from others. In general, given N workers, the worst-case staleness under the R^{2}SP scheme is $N-1$, as opposed to $2(N-1)$ under the SSP-like scheme. We next show in theory that R^{2}SP results in higher statistical efficiency than the SSP scheme.

D. Convergence Analysis

We analyze the convergence speed of R^{2}SP by deriving the *regret bound*, which quantifies how fast the learning accuracy improves as the training iteration proceeds.

Lemma 1: Under the R^{2}SP scheme, let ω_k^i be the parameters after the update from worker-i in iteration k. We have

$$\omega_k^i = \omega_0 + \sum_{k'=1}^{k-1}\sum_{i'=1}^{N} u_{k'}^{i'} + \sum_{i'=1}^{i} u_k^{i'},$$

where ω_0 is the initial parameter, and N is the total number of workers.

In a nutshell, Lemma 1 states that ω_k^i is refined from initial value ω_0 by two groups of updates:

1. Updates from all workers in iterations 1 to $k-1$.

2. Updates from workers $1, 2, ..., i$ in iteration k, which have already been received by PS.

In order to analyze the convergence rate, we rewrite parameter ω_k^i as a sequence indexed by the accumulative update number t, where t is $(k-1)N+i$:

$$\omega_k^i = \omega_t = \omega_0 + \sum_{t'=0}^{t} u_{t'}, \text{ where } u_t := u_{t/N}^{t \bmod N}.$$

Based on Lemma 1, our goal is to find the bound of the *regret* $R[W]$ associated with the parameter sequence $\{\omega_t\}$. The analysis is based on two standard assumptions.

Assumption 1 (L-Lipschitz): For convex[3] function $f_t(\omega)$, the sub-differentials $||\nabla f_t(\omega)||$ are bounded by a constant L.

Assumption 2 (Bounded Diameter): For any ω, ω' in the parameter space, let $D(\omega||\omega') = \frac{1}{2}||\omega - \omega'||^2$ be their distance. We have $D(\omega||\omega') \leq F^2$ for some constant F.

Theorem 1: Define $u_t := -\eta_t \nabla f_t(\omega_t) = -\eta_t g_t$, where $\eta_t = \frac{\sigma}{\sqrt{t}}$ and $\sigma = \frac{F}{2L}$. Under Assumptions 1 and 2, we bound the *regret* by

$$R[W] := \frac{1}{T}\sum_{t=1}^{T}\Big[f_t(\omega_t) - f_t(\omega^*)\Big] \leq \frac{5FL}{2\sqrt{T}}.$$

Before proving Theorem 1, we give the following lemma to help bound the instantaneous regret for a certain iteration.

Lemma 2: For all $t > 0$ and $\omega \in \mathbb{R}^n$, we have:

$$\langle \omega_t - \omega^*, g_t \rangle = \frac{\eta_t}{2}||g_t||^2 + \frac{1}{\eta_t}\Big[D(\omega^*||\omega_t) - D(\omega^*||\omega_{t+1})\Big].$$

Proof: According to Assumption 2, we have

$$D(\omega^*||\omega_{t+1}) - D(\omega^*||\omega_t) = \frac{1}{2}||\omega^* - \omega_t + \omega_t - \omega_{t+1}||^2 - \frac{1}{2}||\omega^* - \omega_t||^2$$
$$= \frac{1}{2}||\omega^* - \omega_t + \eta_t g_t||^2 - \frac{1}{2}||\omega^* - \omega_t||^2 = \frac{1}{2}\eta_t^2||g_t||^2 - \eta_t\langle \omega_t - \omega^*, g_t\rangle.$$

Moving $\langle \omega_t - \omega^*, g_t\rangle$ to the left, we prove the lemma. ■

We are now ready to prove Theorem 1.

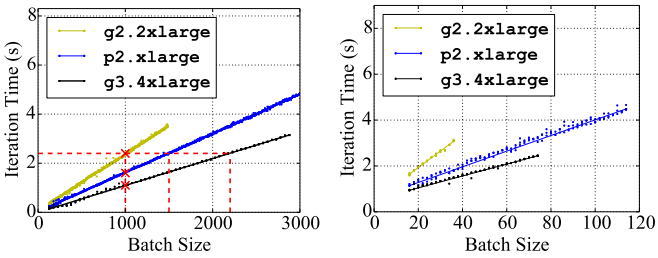
Proof: Because $f_t(\omega)$ is convex, we use Lemma 2 to expand $R[W]$ to get its upper bound:

$$R[X] = \frac{1}{T}\sum_{t=1}^{T}[f_t(\omega_t) - f_t(\omega^*)] \leq \frac{1}{T}\sum_{t=1}^{T}\langle \omega_t - \omega^*, g_t\rangle$$
$$= \frac{1}{T}\sum_{t=1}^{T}\Big[\frac{\eta_t}{2}||g_t||^2 + \frac{D(\omega^*||\omega_t) - D(\omega^*||\omega_{t+1})}{\eta_t}\Big] = \frac{1}{T}\sum_{t=1}^{T}\frac{\eta_t}{2}||g_t||^2$$
$$+ \frac{1}{T}\Big[\frac{D(\omega^*||\omega_1)}{\eta_1} - \frac{D(\omega^*||\omega_{T+1})}{\eta_T} + \sum_{t=2}^{T} D(\omega^*||\omega_t)\Big(\frac{1}{\eta_t} - \frac{1}{\eta_{t-1}}\Big)\Big].$$

We then bound each term in the equation above. The first term can be bounded using the L-Lipschitz property in Assumption 1:

$$\sum_{t=1}^{T}\frac{\eta_t}{2}||g_t||^2 \leq \sum_{t=1}^{T}\frac{\eta_t}{2}L^2 \leq \sum_{t=1}^{T}\frac{\sigma L^2}{2\sqrt{t}} \leq \int_0^T \frac{\sigma L^2}{2\sqrt{t}} = \sigma L^2\sqrt{T}.$$

[3]Strictly speaking, most DL problems are non-convex. But in current DL community, by adopting the SGD aglorithm, they are treated as being convex. The resultant *local minimum*, avoiding over-fitting, can usually work well.



(a) ResNet32 on CIFAR-10 dataset (b) Inception-v3 on ImageNet dataset

Fig. 5: Relationship between iteration time and batch size for different models and different EC2 instance types.

With the bounded diameter property in Assumption 2, we have

$$\frac{D(\omega^*||\omega_1)}{\eta_1} - \frac{D(\omega^*||\omega_{T+1})}{\eta_T} + \sum_{t=2}^{T} D(\omega^*||\omega_t)\Big(\frac{1}{\eta_t} - \frac{1}{\eta_{t-1}}\Big)$$
$$\leq \frac{F^2}{\sigma} - 0 + \frac{F^2}{\sigma}\sum_{t=2}^{T}\Big[\sqrt{t} - \sqrt{t-1}\Big] = \frac{F^2}{\sigma}\sqrt{T}.$$

Putting it altogether, we have:

$$R[W] \leq \frac{\sigma L^2}{\sqrt{T}} + \frac{F^2}{\sigma\sqrt{T}} = \frac{FL}{2\sqrt{T}} + \frac{2F^2}{\sqrt{T}F/L} = \frac{5FL}{2\sqrt{T}}.$$

This completes the proof of Theorem 1. ■

Remarks. Theorem 1 further indicates that the convergence speed under R^{2}SP is faster than that under SSP. Following the same setup, the regret bound under SSP, according to the analysis in [15], is $4FL\sqrt{2(b+1)/T}$, in which b is the aforementioned *bound on progress gap*. Even if we set b to the minimum value 1, the regret bound under SSP ($8FL/\sqrt{T}$) remains larger than that under R^{2}SP. We will empirically confirm such superiority of R^{2}SP in Sec. VI.

IV. Batch Size Tuning

So far, our discussions were limited to homogeneous clusters where all GPU workers have the same computing capability. While this is usually the case in practice [6]–[9], it is not uncommon that many DL models were trained in *heterogeneous* clusters with different generations of GPUs [17]–[19]. For example, a budget-limited user might use a combination of whatever GPU instances it can find in the EC2 spot market offering high performance-cost ratio, regardless of their types.

Resource wastage. In such heterogeneous clusters, when applying our R^{2}SP in the distributed DL process, one problem is the *resource wastage*. This is because existing DL frameworks [7], [8] enforce a *uniform batch size* across workers. As GPUs are of different computing capabilities, their batch processing time can be highly divergent. This is confirmed in Fig. 5, where we measured the iteration time against batch size for ResNet32 and Inception-v3 using different EC2 GPU instances. As shown in Fig. 5a, different instances take different time to iterate over a batch of 1000 samples.

As workers now have different iteration time T, the inter-update gap (T/N in Sec. III-B) is determined by the slowest worker (i.e., the one with the longest T). Consequently, fast

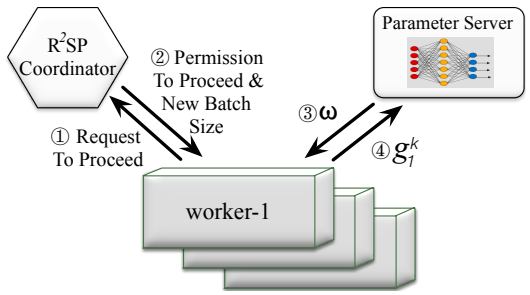


Fig. 6: Architecture and workflow of R²SP-Coordinator (in iteration k). The circled numbers represent the execution order.

workers have to wait for their turn to come as they finish iterations earlier, hence wasting their computing cycles and impairing the learning efficiency.

Batch size tuning. To address this problem, we propose *batch size tuning* for heterogeneous clusters. Our key insight is to *adaptively tune each worker's batch size based on its computing capability, so as to equalize their iteration time*. That is to say, the *faster* a worker is, the *larger* batch size it shall have. Referring back to Fig. 5a, suppose initially the batch size of the three instances is 1000. As `p2.xlarge` and `g3.4xlarge` instances are faster than `g2.2xlarge`, we respectively increase their batch sizes to around 1500 and 2200, so that all three instances finish an iteration with similar time costs.

Batch size tuning is designed based on the *linear* relationship between a worker's batch size and its iteration time, as also revealed in Fig. 5. That linear relationship represents the *speed* that training samples are processed, which is stable for a GPU instance when training a given DL model. Therefore, given the potential idle time of a fast worker and its sample processing speed, we can easily work out how to tune its batch size to keep it busy during training without delaying others.

Moreover, when incorporating batch size tuning into R²SP, we adopt a *linear scaling rule* on the learning rate, similar to [12]. That is, after we increase a worker's batch size, we also proportionally scale up its learning rate when refining the model with that worker's gradient. This way, we can guarantee that each sample has the *same level of influence* on model refining so that biased results can be avoided.

Intuitively, for well-shuffled *i.i.d.* (i.e., independent and identically distributed) data samples, we can expect improved learning efficiency with batch size tuning. This is because it enables more samples processed by workers in each iteration, and under the *i.i.d.* condition, each sample is expected to have a similar contribution to model convergence. We shall illustrate this benefit through empirical studies in Sec. VI-C.

V. Prototype Implementation

We have implemented R²SP as a ready-to-use Python library, called `R²SP-Coordinator`, for popular DL frameworks such as TensorFlow [7] and MXNet [8].

Workflow. Fig. 6 illustrates how `R²SP-Coordinator` can be used to coordinate workers. It implements the main logic of the R²SP scheme by controlling when a given worker can proceed

Algorithm 1 Workflow with `R²SP-Coordinator`

Worker: i=1, 2, ..., N:

procedure WorkerIterate(k)
  pull latest parameter ω from PS
  load sample batch I_k^i
  calculate local gradient $g_k^i = \frac{1}{|I_k^i|}\sum_{s\in I_k^i}\nabla f(s,\omega)$
  $u_k^i \leftarrow -\frac{|I_k^i|}{I_0^i}\eta_k g_k^i$, push u_k^i to PS
    ▷ the learning rate is linearly scaled with the worker batch size
  send `Request-to-Proceed` signal to `R²SP-Coordinator`
  block before receiving the `Permission-to-Proceed` signal and the tuned batch size $|I_{k+1}^i|$

Parameter Server (PS):

procedure ParameterServerIterate
  update parameters $\omega \leftarrow \omega + u_k^i$

R²SP-Coordinator:

procedure R²SP-CoordinatorIterate
  adjust batch sizes of the faster workers (if any) to eliminate their resource wastage
  issue `Permission-to-Proceed` signal to appropriate worker at appropriate time

to the next iteration. In particular, before entering the next iteration, ① a worker first sends a `Request-to-Proceed` signal to the `R²SP-Coordinator` and waits for its permission to proceed. The `R²SP-Coordinator`, on the other hand, determines which worker has the next turn and the earliest time for that worker to proceed, so as to maintain the worker update order (Sec. III-C) and the inter-update gap (Sec. III-B). Once it comes to the worker's turn, ② `R²SP-Coordinator` notifies the worker by issuing a `Permission-to-Proceed` signal. Upon receiving the permission, the worker proceeds to the next iteration. It ③ pulls the latest parameters from the PS and ④ pushes back the update after the training iteration has completed. Algorithm 1 summarizes the entire workflow.

Batch size tuning. To enable *batch size tuning* for heterogeneous clusters, we piggyback the tuned batch size on the `Permission-to-Proceed` signal. This allows faster workers to process more samples in an iteration without being idle or delaying the slower workers.

Profiling iteration time. To determine the temporal gap between worker updates, the `R²SP-Coordinator` requires to know the iteration time T. We use the EMA (Exponential Moving Average) method to accurately learn the value of T.

Dealing with Fluctuation. So far, we have assumed that the iteration time on a worker is *stable* across iterations. While this generally holds *in expectation*, in practice the exact time for each individual iteration could *slightly fluctuate* around the expected value, mainly due to random perturbations of GPU processing speed and network state. In this case, strictly enforcing the inter-update gap would block those workers that finish their iterations earlier than expected. Consequently, the *overall* iteration time gets prolonged.

To remain robust to such random variations, we employ a *relaxation factor* r in $[0, 1]$. That is, given N workers and their

iteration time T, we relax the update gap from T/N to rT/N. This allows early birds to proceed to the next iteration, without affecting late arrivals. In our implementation, we set $r = 0.8$ and find it to yield good performance (details in Sec. VI-D).

Overhead. In our implementation, we use Apache Thrift [29], a light-weight RPC protocol developed by FaceBook, as the underlying communication protocol between the coordinator and workers. Given the small amount of control signals exchanged (only a few bytes for each signal) and the low computational complexity of the R^{2}SP algorithm, the overhead of our implementation is negligible.

VI. Evaluation

In this section, we evaluate R^{2}SP in two EC2 clusters with homogeneous and heterogeneous GPU workers, respectively. We first demonstrate the optimality of R^{2}SP in mitigating network contentions for fast DL training, and then evaluate the effectiveness of *batch size tuning* in a heterogeneous cluster. Finally, we perform sensitivity analysis to identify potential factors that may affect the performance of R^{2}SP.

A. Experimental Setup

EC2 deployment. We deployed two GPU clusters in Amazon EC2. The first cluster (**Cluster-A**) consists of 4 PS nodes and 16 *homogeneous* workers interconnected by 10 Gbps links. Each PS node is a `c5.9xlarge` instance with 36 vCPUs and 72 GB RAM; each worker is a `g3.4xlarge` instance equipped with an NVIDIA Tesla M60 GPU. We used cluster-A for performance comparison between R^{2}SP and existing synchronization schemes, namely BSP, ASP and SSP.[4]

The second cluster (**Cluster-B**) consists of 2 PS nodes (`c5.9xlarge`) and 8 *heterogeneous* GPU workers including two `g2.2xlarge` instances (NVIDIA GRID K520), two `p2.xlarge` instances (NVIDIA K80), and four `g3.4xlarge` instances (NVIDIA Tesla M60). We used cluster-B for evaluating the benefits of batch size tuning in heterogeneous settings.

Datasets and Models. We trained two typical image classification datasets using TensorFlow [7]: (1) CIFAR-10 [25] and (2) ILSVRC12. The latter is a subset [26] of ImageNet22K containing 1.28 million of training images in 1000 categories. In particular, we trained ResNet32 [30], AlexNet [2] and VGG16 [31] models over the CIFAR-10 dataset, and trained Inception-v3 model [32] over the ILSVRC12 dataset. The default batch size, unless otherwise specified, is set to 128 for CIFAR-10 dataset, and 32 for ILSVRC12 dataset. Meanwhile, for models trained upon CIFAR-10 dataset, the initial learning rate is set to 0.01, while for ILSVRC12 it is 0.045.

Metrics. We trained a DL model until it converges and used the training time as a metric of training efficiency. In our evaluation, the training is deemed to *converge* if the loss value, in 10 consecutive iterations, falls below 0.25 for ResNet32 (loss reduced by around 90%), 0.5 for AlexNet (loss reduced by around 80%), 2.3 for VGG16 (loss reduced by around 30%), and 6.0 for Inception-v3 (loss reduced by around 60%). The training efficiency is further broken down into *statistical efficiency* and *hardware efficiency*. The former is measured by the average number of iterations performed by each worker towards convergence; the latter is measured by the mean iteration time across workers during the training process.

[4]By default, TensorFlow does not support the SSP scheme. We implemented it with a `worker-coordinator` that enforces fast workers to wait if the bound of progress gap—set to 5 iterations in our experiment—is reached.

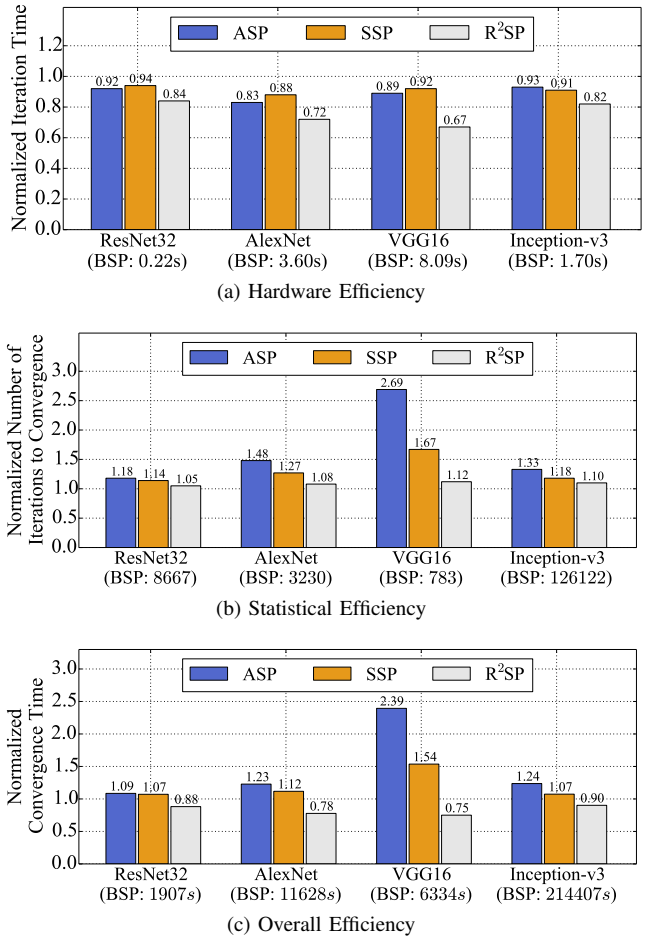


Fig. 7: Performance comparison among BSP, ASP, SSP and R^{2}SP. The presented values are normalized by that under BSP.

B. Benefits in Homogeneous Cluster

Efficiency. We evaluate the efficiency of R^{2}SP against the three existing synchronization schemes in cluster-A. Fig. 7 compares their performance, where we use BSP as the baseline and normalize the results of the other schemes by that of BSP.

We start with hardware efficiency. Fig. 7a shows that relaxed synchronization (i.e., ASP, SSP and R^{2}SP) results in faster training iteration than that with the BSP scheme, as communication bottleneck is less of a concern. Among all four schemes, R^{2}SP minimizes the network contention and hence attains the *highest* hardware efficiency. Notably, for network-intensive models such as AlexNet and VGG16 (Table II-B), R^{2}SP speeds up their iterations by around 30% than BSP.

We next turn to statistical efficiency. Fig. 7b shows that compared with BSP, more iterations are needed for convergence

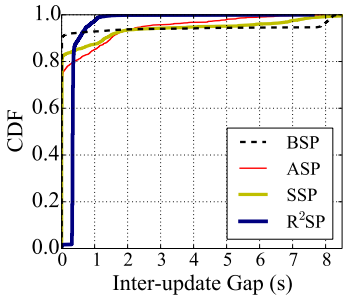


Fig. 8: CDF of the inter-update gap time when training VGG16.

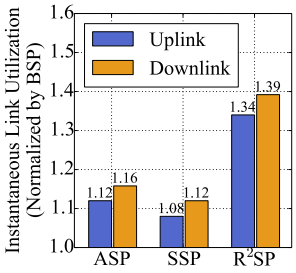


Fig. 9: Instantaneous PS bandwidth utilized by each worker during transferring.

Instance Type	g2.2x	p2.x	g3.4x
Blocking Time w/o Tuning (s)	0	0.62	0.82
Speed (sample/s)	429	628	917
Tuned Batch Size	512	901	1264

TABLE II: Fast workers are blocked long by vanilla R^{2}SP. Batch size tuning eliminates such idle periods.

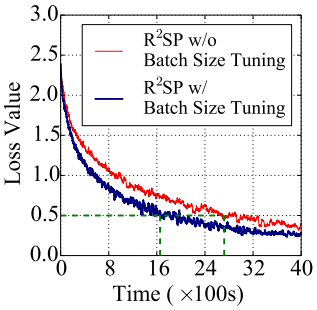


Fig. 10: Convergence curves of ResNet32 in Cluster-B.

with relaxed synchronization—a consequence that stale parameters harm statistical efficiency. Nevertheless, R^{2}SP results in the minimum loss among the three asynchronous schemes, as it bounds the staleness by a minimum amount by enforcing a fixed round-robin order to worker updates (Sec. III-C).

Fig. 7c shows the overall efficiency. Despite the slight loss of statistical efficiency, R^{2}SP remains the fastest in convergence due to its salient speedup in iterations. Specifically, it outperforms BSP by up to 25% (VGG16). Note that in our experiments, BSP appears more efficient than ASP and SSP, which is consistent with previous observations [10]–[12].

Contention mitigation. To further illustrate how R^{2}SP helps mitigate network contention, we resort to deep-dive measurements. We use the *inter-update gap* being *zero* as an indicator of network contention: when two updates are made simultaneously, the gap in between is zero. Therefore, the more zero gaps, the more severe the network contention. Fig. 8 depicts the distribution of inter-update gaps measured under the four schemes when training VGG16. As expected, BSP results in most zero gaps: zero gaps account for $15/16$ of the total number; the remaining $1/16$ span the entire iteration (~ 8 s). This is because cluster-A has 16 workers, all making updates simultaneously in front of the end-of-iteration barriers imposed by BSP. ASP and SSP schemes, while slightly better than BSP, remain *suboptimal* in contention mitigation, under which zero gaps account for 75% and 83%, respectively. In comparison, with R^{2}SP, almost all inter-update gaps are equal to $1/16$ of the iteration span, meaning that the updates are evenly staggered with minimum contentions in between.

We further measured the mean bandwidth at which workers communicate with the PS on its uplink and downlink under the four schemes. We depict the results, normalized by that of BSP, in Fig. 9. As R^{2}SP minimizes network contention, each worker ends up with much higher instantaneous bandwidth than that with the other schemes, which translates to accelerated iterations as shown in Fig. 7a.

C. Batch Size Tuning in Heterogeneous Clusters

We now evaluate the effectiveness of batch size tuning in the heterogeneous environments. We trained the ResNet32 model in cluster-B using R^{2}SP, with and without batch size tuning. For each worker, we set the initial batch size to 512. As shown in Table II, with equal batch size, fast workers (i.e., `p2.xlarge` and `g3.4xlarge` instances) complete an iteration earlier and have to wait for their turn to proceed to the next iteration, resulting in salient blocking time. With batch size tuning, we assign larger batches to `p2.xlarge` and `g3.4xlarge` instances in proportion to their processing speed. This would keep fast workers busy in an iteration, hence avoiding resource wastage due to idling instances.

Fig. 10 compares the convergence curve of training ResNet32 model using R^{2}SP with (blue) and without batch size tuning (red). The former leads to faster convergence. In particular, given the convergence criterion specified in Sec. VI-A, batch size tuning itself can accelerate convergence by around 40%.

D. Sensitivity Analysis

Finally, we perform sensitivity analysis on the factors that may affect the performance of R^{2}SP. Specifically, we evaluate the behaviors of R^{2}SP with various link bandwidth and values of the *relaxation factor* (Sec. V).

Link bandwidth. Our previous evaluations were based on 10 Gbps network. To quantify how R^{2}SP performs when network is a even more severe bottleneck, we trained AlexNet and VGG16 models in cluster-A with *throttled* link bandwidth. Fig. 11 shows the iteration speedup of R^{2}SP over BSP with different bandwidth. We observe the same trend for both models: the *more severe* the network bottleneck is, the *more significant speedup* R^{2}SP can provide. R^{2}SP therefore arises as a promising solution for network-intensive training, such as learning at an extremely large scale [12] and geo-distributed machine learning [22].

Relaxation factor. Recall that in Sec. V, we have introduced a *relaxation factor* to avoid blocking early-bird workers in the presence of iteration time fluctuations. To evaluate its effectiveness, we trained VGG16 model in cluster-A (for 200 iterations) with varying relaxation factors from 0 to 1. In each experiment, we measured the average *blocking time* per iteration, together with the *mean iteration time*. Fig. 12 shows the results. As the relaxation factor gets smaller, the mean blocking time decreases. However, configuring a smaller relaxation factor means that worker updates are more loosely staggered, which, in turn, adds the risk of network contention and results in prolonged iteration time. Owing to this tradeoff, we see in Fig. 12 that the sweet spot for the relaxation factor is achieved at 0.8, which leads to the shortest iteration.

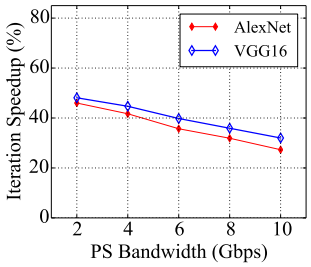


Fig. 11: The more severe the network bottleneck, the larger benefit (vs. BSP) from R^{2}SP.

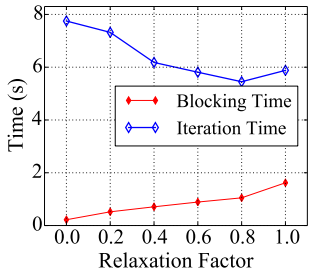


Fig. 12: Blocking time and iteration time when using different relaxation factors under R^{2}SP.

VII. Related Work

Mitigating Network Bottleneck. Network bottleneck has long plagued distributed deep learning, and many approaches have been developed to address that problem. Notably, some works have proposed to compress the DL models—by *quantizing* each model parameter to fewer bits [33], or by *pruning* the insignificant connections [34]—to sacrifice accuracy for reduced worker-PS transmission amount. Meanwhile, parameters (basically a matrix) for the *fully-connected* layers in neural networks can be decoupled into two vectors, and it has been observed that transferring those vectors rather than the initial parameter matrices is more communication-efficient [13]. Furthermore, Gaia [22] has recently proposed to transfer *merely* those gradients *significant* enough for model refining, so as to reduce the communication frequency over bottlenecked links.

Nonetheless, those approaches are orthogonal to R^{2}SP: they focus *only* on reducing the total transmission demand, and *fail* to notice that workers can *time-multiplex* the bottlenecked links to have more *instantaneous* bandwidth for fast transmission.

Efficient Learning in Heterogeneous Clusters. For efficient deep learning in heterogeneous clusters (Sec. IV), Zhang *et al.* [18] and Jiang *et al.* [17] seek for improved statistical efficiency by dynamically adjusting the learning rate based on the temporal extent of parameter staleness, but they cannot eliminate resource wastage elaborated in Sec. IV. Further, FlexRR [35] seeks to avoid such resource wastage by offloading some training work from the slower workers to the faster ones, but this would incur much more communication overhead. Those solutions are thus less suitable than batch size tuning.

VIII. Conclusion

In this work, we have designed the round-robin synchronous parallel (R^{2}SP) scheme to mitigate the communication bottlenecks in PS-based learning systems. R^{2}SP evenly staggers the worker-PS communications within iterations for improved hardware efficiency, while in the meantime enforcing a fixed round-robin update order to bound the staleness of parameters by a minimum amount. We have also extended R^{2}SP to heterogeneous clusters and implemented it as a Python library for prevalent learning frameworks. Prototype evaluations show that R^{2}SP can speed up model convergence by up to 25%.

Acknowledgement

The research was supported in part by RGC GRF grants under the contracts 16206417, 16207818 and 26213818, as well as an RGC CRF grant under the contract C7036-15G. We also thank Huangshi Tian for his assistance in the evaluations.

References

[1] Y. LeCun et al. Deep learning. *Nature*, 521(7553):436–444, 2015.
[2] A. Krizhevsky et al. ImageNet classification with deep convolutional neural networks. In *NIPS*. 2012.
[3] C. Farabet et al. Learning hierarchical features for scene labeling. *IEEE Trans. Pattern Anal. Mach. Intell.*, 35(8):1915–1929, 2013.
[4] R. Collobert et al. Natural language processing (almost) from scratch. *J. Mach. Learn. Res.*, 12(Aug):2493–2537, 2011.
[5] I. Sutskever et al. Sequence to sequence learning with neural networks. In *NIPS*. 2014.
[6] M. Li et al. Scaling distributed machine learning with the parameter server. In *USENIX OSDI*. 2014.
[7] M. Abadi et al. TensorFlow: A system for large-scale machine learning. In *USENIX OSDI*. 2016.
[8] T. Chen et al. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv:1512.01274*, 2015.
[9] A. Paszke et al. Automatic differentiation in pytorch. In *NIPS-W*. 2017.
[10] H. Cui et al. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *ACM Eurosys*. 2016.
[11] J. Chen et al. Revisiting distributed synchronous sgd. *arXiv preprint arXiv:1604.00981*, 2016.
[12] P. Goyal et al. Accurate, large minibatch sgd: training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.
[13] H. Zhang et al. Poseidon: An efficient communication architecture for distributed deep learning on gpu clusters. In *USENIX ATC*. 2017.
[14] S. Shi et al. MG-WFBP: Efficient data communication for distributed synchronous sgd algorithms. In *IEEE INFOCOM*. 2019.
[15] Q. Ho et al. More effective distributed ml via a stale synchronous parallel parameter server. In *NIPS*. 2013.
[16] J. Langford et al. Slow learners are fast. In *NIPS*. 2009.
[17] J. Jiang et al. Heterogeneity-aware distributed parameter servers. In *ACM SIGMOD*. 2017.
[18] W. Zhang et al. Staleness-aware async-SGD for distributed deep learning. *arXiv preprint arXiv:1511.05950*, 2015.
[19] F. Song et al. A scalable framework for heterogeneous GPU-based clusters. In *ACM SPAA*. 2012.
[20] J. Dean et al. Large scale distributed deep networks. In *NIPS*. 2012.
[21] M. Li et al. Efficient mini-batch training for stochastic optimization. In *ACM KDD*. 2014.
[22] K. Hsieh et al. Gaia: Geo-distributed machine learning approaching LAN speeds. In *USENIX NSDI*. 2017.
[23] H. Cui et al. Exploiting iterative-ness for parallel ml computations. In *ACM SoCC*. 2014.
[24] T. M. Chilimbi et al. Project Adam: Building an efficient and scalable deep learning training system. In *USENIX OSDI*. 2014.
[25] A. Krizhevsky et al. Learning multiple layers of features from tiny images. 2009.
[26] J. Deng et al. ImageNet: A large-scale hierarchical image database. In *IEEE CVPR*. 2009.
[27] Cisco devices only support full-duplex. https://www.cisco.com/c/en/us/support/docs/lan-switching/ethernet/10561-3.html.
[28] D. Halperin et al. Augmenting data center networks with multi-gigabit wireless links. In *ACM SIGCOMM*. 2011.
[29] Apache Thrift. https://thrift.apache.org/.
[30] K. He et al. Deep residual learning for image recognition. In *IEEE CVPR*. 2016.
[31] K. Simonyan et al. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
[32] TensorFlow Inception-V3. https://github.com/tensorflow/models/tree/master/research/inception.
[33] Y. Gong et al. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.
[34] S. Han et al. Learning both weights and connections for efficient neural network. In *NIPS*. 2015.
[35] A. Harlap et al. Addressing the straggler problem for iterative convergent parallel ml. In *ACM SoCC*. 2016.