Mitigating Server-side Communication Bottlenecks in Distributed Learning with Round-Robin Participant Coordination

Jiayi Zhang, *Student Member, IEEE,* Chen Chen, *Member, IEEE,* Zuo Gan, *Student Member, IEEE,* Wei Wang, *Member, IEEE,* Bo Li, *Fellow, IEEE,* and Minyi Guo, *Fellow, IEEE*

***Abstract*—Deep neural networks are increasingly trained in a distributed manner—in either clusters or with federated devices, where the participants jointly refine the global model with their gradients calculated locally. More often than not, those gradients are collected to a central server in a synchronous manner to avoid the negative impact of stale updates. However, when all the participants communicate their gradients to the server in such a uniform pace, the network on the server side—under intense contention—often becomes a performance bottleneck. To address this problem, for the cluster environment we propose the *Round-Robin Synchronous Parallel* (R^2SP) scheme, which coordinates the participants to make updates in an *evenly-gapped, round-robin* manner. This way, we can minimize the network contention with a minimum cost of the update quality; we also propose to incorporate adaptive batch sizing in R^2SP to address the hardware heterogeneity among workers. Moreover, for the federated learning (FL) scenarios, we note that it is necessary yet challenging to apply the insight of R^2SP to mitigate the network bottleneck in the FL server, given that there are a huge number of participants with unstable resources and inconsistent data distributions. To tackle those challenges, we further propose FL-R^2SP, which extents the coordination units from individual participants to participant *groups*—with the resource instability and data heterogeneity tackled within each group. We have implemented R^2SP and FL-R^2SP respectively with TensorFlow and PyTorch, and extensive EC2 experiments show that R^2SP and FL-R^2SP can respectively speed up model convergence for clustered and federated scenarios by over 20%.**

***Index Terms*—Distributed Machine Learning, Synchronization, Network Bottleneck, Federated Learning**

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]. With the ever increasing data volume and model complexity, DL models are overwhelmingly trained in a distributed manner (in a *dedicated cluster* [6] or with a set of *federated devices* [7]). Parameter Server (PS) is a typical architecture for distributed model training [8], [9], [10], [11], where a number of parallel participants *iteratively* compute their local gradients and refine the global model via remote communication.

Regarding the coordination of different participants, Bulk Synchronous Parallel (BSP) is a common scheme adopted in many model training practices [12], [13], [14]. Under BSP, participants synchronize with the server at the same pace: they will not proceed until the global model is fully refined with 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.

Under the BSP scheme, *network contention* arises as a main source of the communication bottleneck. In distributed machine learning systems with *homogeneous* devices, workers synchronize at *roughly the same time*, contending for the network bandwidth of the server. Consequently, each worker has to bear 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 times.

A naive implementation of this approach goes to the ASynchronous Parallel (ASP) scheme [13], [15], where each worker communicates with the server *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* contention mitigation effect. 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 [16], [17], [18] (i.e., low *statistical efficiency*).

In this paper, our goal is to (1) *minimize* the network contention in distributed machine learning systems 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 participant-coordinating scheme—*Round-Robin Synchronous Parallel* (R^2SP). Under R^2SP, workers make updates to the server 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 server 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 clusters with *heterogeneous* GPUs—a possible case in reality due to local inventory shortage or improper resource allocation [18], [19], [20], [21]. Owing to such heterogeneity, those workers would possess diverse computing capabilities. Directly applying R^2SP can be *inefficient*: fast workers finish computations

Jiayi Zhang, Chen Chen, Zuo Gan and Minyi Guo are with the School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University (email: jiayizhang@sjtu.edu.cn; chen-chen@sjtu.edu.cn; gz944367214@sjtu.edu.cn; myguo@sjtu.edu.cn).

Wei Wang and Bo Li are with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology (email: weiwa@cse.ust.hk; bli@cse.ust.hk).

Chen Chen is the corresponding author.

quickly, but have to wait long for their turn to make update to the server. 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 clients now do more useful work without idling under the R^{2}SP scheme, the model convergence speed can be further accelerated.

While R^{2}SP is primarily proposed for cluster environment, we notice that the basic idea of R^{2}SP is also needed by Federated Learning (FL) [22], [23], which has recently become a very hot research topic. In many real-world scenarios, the training samples cannot be accessed out of their original locations (i.e., cellphones or hospitals); FL is thus proposed as a privacy-preserving training scheme that allows edge devices to collaborate without disclosing their private data. In particular, FL also employs a PS-like architecture, with a FL server that collects the updates from edge devices. Meanwhile, similar to the PS architecture, most FL practices employ the BSP scheme for client coordination [23], [24], [25], [26]. As reported by Google [26], the FL server for realistic applications may need to deal with a population size of hundreds of millions; in that case, the network on the FL server may also become a performance bottleneck [27], [28], [29].

However, it turns out to be a non-trivial task to apply R^{2}SP in FL scenarios. R^{2}SP requires that all the participants be well controlled—each participant under R^{2}SP shall push and pull the model updates at the dictated time. However, in FL scenarios, the available resources on client devices may fluctuate drastically, making their behaviors highly unpredictable; worse, some clients may even lose connection in the middle. Hence, it is actually impossible to make update coordination at the client granularity. Moreover, another plaguing problem of FL is data heterogeneity: contrary to cluster environment where data is well shuffled across different workers, in FL scenarios the local datasets of some clients may be highly skewed, being poisonous for global model convergence.

To address these challenges, we customize the vanilla R^{2}SP strategy for FL scenarios and propose *Federated Learning Round-Robin Synchronous Parallel*, or FL-R^{2}SP. The primary insight of FL-R^{2}SP is to extent the coordination granularity from individual *clients* to client *groups*. FL-R^{2}SP randomly binds each FL client to a group: within a group, client updates are aggregated in a synchronous manner as in BSP; yet across different groups, such aggregations are triggered in a round-robin manner as in R^{2}SP. This way, we can evenly multiplex the server-side network usage despite unstable client participation. FL-R^{2}SP further deals with the stragglers and data outliers within each group—by enabling fractional gradient aggregation and filtering out the poisonous gradients with cosine-similarity analysis.

We respectively implement R^{2}SP and FL-R^{2}SP atop TensorFlow [9] and PyTorch [30]. For R^{2}SP, EC2 deployment atop 16 GPU-workers (`g3.4xlarge` instances) demonstrates that it can reduce the iteration time by more than 30%, with an overall convergence speed up of 25%. Meanwhile, with batch size tuning, R^{2}SP can further speed up model convergence by 40% in a heterogeneous cluster. Regarding FL-R^{2}SP, evaluations in an emulated FL setup with 128 clients show that, FL-R^{2}SP can remarkably enhance the server-side bandwidth utilization, reducing the average communication time by over 86% and speeding up model convergence by over 20%.

II. Research Background

A. Distributed Training in Cluster or Federated Scenarios

Model training is a crucial process for AI innovations. Given a neural network model, the goal of the training process is to find the model parameters that minimizes the loss function over the entire training dataset. Mini-batch Stochastic Gradient Descent [31], [32], or simply SGD, is the state-of-the-art algorithm to train model parameters. Its basic idea is to *iteratively* refine the model parameter with the worker updates.

Many real-world deep learning tasks have very large datasets, and the deep neural networks are usually trained in a cluster of machines. Nowadays, two typical cluster architectures for distributed model training are all-reduce and parameter server (PS). In all-reduce architecture [33], [14], all the participants exchange their local gradients in a peer-to-peer manner, which is prone to single point of failure; in contrast, PS architecture [8], [9], [10], [11] is more robust and flexible, which is the focus of this paper. 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 *pulls* the latest model parameters from the PS, computes the local update (gradient) based on its sample batch, and finally *pushes* the gradient to the PS to refine the parameters. Regarding the pace control of different participants, BSP (Bulk Synchronous Parallel) is the most commonly adopted synchronization scheme. It imposes a *synchronization barrier*, with which workers cannot proceed to the next iteration before the parameters are fully updated.

Apart from distributed learning in cluster environments, where the ML practitioners have full control over the training data, in many real-world scenarios, the training samples are privacy-sensitive and must be kept where they were generated. To train models without centralizing such private data, an increasingly popular technique is FL [22], [23]. Under FL, clients compute model updates locally and there is a central server that periodically aggregate their updates, typically also in a synchronous[1] manner as in the PS architecture.

B. Communication Bottlenecks in Distributed Training

While BSP is commonly adopted for distributed model training, a severe performance problem it suffers is the communication bottleneck at synchronization time.

Empirical study on communication bottlenecks in cluster environment. In cluster environments, as the training in each iteration gets dramatic speedup (usually in sub-seconds) by

[1]Recently, asynchronous client coordination is also adopted by some research works [34], [35], [36], yet it is prone to stale updates (yielding low accuracy as confirmed in our evaluations in Sec. VI-B2), and usually requires additional client control mechanisms for remediation. Moreover, asynchronous aggregation is not friendly to encryption (for example, a premier privacy enhancing technology—*secure aggregation* [37]—is not supported in asynchronous mode). To date, BSP is still the mainstream client coordinating scheme for FL [26], [38], [23], [24], [25], which is the focus of this paper.

	ResNet32	AlexNet	VGG16	Inception-v3
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 when training DL models in a cluster with 8 GPU-workers.

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[2] 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 [9] using the BSP scheme: ResNet32, AlexNet, VGG16, and Inception-v3. We used CIFAR-10 [39] datasets for the first three models and ImageNet [40] 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.

Empirical study on communication bottlenecks in federated environment. For federated learning scenarios, we have also observed similar performance degradation. In typical FL scenarios [26], [35], [38], there is a large number of low-end devices concurrently interacting with the FL server. While many existing works [41], [42], [43] focus on mitigating the transmission delay for the FL *clients*, we find that the FL *server* may also be a performance bottleneck due to intense network contention. As reported by industrial FL practitioners like Google [26] and Meta [35], FL often involves up to millions of clients; although the model size for FL is usually smaller than that in clusters, such a huge client quantity would easily incur severe network contention, especially when the server *pushes back* the latest model parameters to all the selected clients simultaneously.

To verify the existence of server-side communication bottlenecks in FL, we further resort to empirical measurements in a EC2 cluster emulating realistic FL setups. That cluster contains 128 `t2.small` instances as FL clients and 1 `m5.4xlarge` instance as the FL server (with an ingress bandwidth of up to 10Gbps). With that cluster we train three models under the BSP scheme: CNN, LSTM and ResNet20, and each client reports its update to the FL server once after every 50 local iterations (please refer to Sec. VI-B for detailed information of the models, datasets and other hyper-parameters). We train each model for 2000 seconds and measure the average communication time in each round (with synchronization waiting time excluded), which is compared with the ideal case with only one client (i.e., without server-side network contention). As shown in Table II, when competing with other clients for the server-side network bandwidth, we can witness a slowdown of up to $34\times$. Such results confirm the need to mitigate server-side communication bottlenecks for FL.

[2]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*.

	CNN	LSTM	ResNet20
Communication-Time w/ Contention	$1.02s$	0.43	$1.98s$
Communication-Time w/o Contention	$0.03s$	$0.03s$	$0.13s$
Communication Slowdown	$34.0\times$	$14.3\times$	$15.2\times$

TABLE II: When training 3 models with 128 clients and 1 FL sever (10Gbps link), the average client communication time is slowed down of up to $34\times$ compared to the case without server-side network contention (i.e., with only 1 client).

C. Prior Solutions and Their Limitations

To mitigate the network bottleneck under BSP, several additional synchronization schemes, like Asynchronous Parallel (ASP) and Stale Synchronous Parallel (SSP), have also been proposed. ASP [13], [15] 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 [16], [44] comes as a middle ground between BSP and ASP. It allows asynchronous training with stale parameters, provided that the *progress gap* between the fastest worker and the slowest is within a bounded amount. While ASP and SSP can mitigate the network contention to some extent, they however incur the problem of stale parameters, which would require more iterations for the model to converge. Besides, SAFA [45] comes as a semi-synchronous scheme, which enforces rigid synchronization within a participant subset and allows asynchronous updating for the other participants; those asynchronous updates are selectively collected based on the actual staleness level. However, it does not fundamentally eliminate the problem of network contention and stale parameters; more iterations would be needed for convergence. Moreover, no methods can *minimize* the bandwidth contentions: without explicit coordination, workers may still have communication collisions 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 later sections.

III. Motivation

A. Network Contention Deep Dive

To understand how the communication bottleneck is formed under the BSP scheme and why it is so dominant, we resort

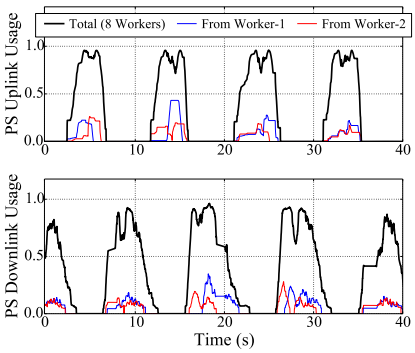


Fig. 1: Total and worker-1/2-originated bandwidth utilization of the PS's downlink and uplink (VGG16).

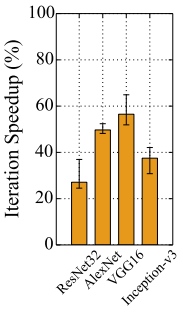


Fig. 2: Iteration speedup if the network contention is eliminated.

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 [46], [47], 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, the uplink/downlink is wasted in idle for more than 50% of iteration time.

In summary, the synchronization barrier imposed by the BSP scheme results in intense network contentions along with low bandwidth utilization.

B. Insights

Intuitively, if the network contention under BSP can be mitigated, 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.[3] This promising result motivates us to explore ways to mitigate network contentions via relaxed synchronization; with relaxed synchronization, it is possible that we minimize the network bottleneck with a slight staleness level.

To summarize, 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.

IV. Round-Robin Synchronization

In this section, we present *Round-Robin Synchronous Parallel* (R^{2}SP) scheme, for easing the communication bottleneck at a minimum cost of statistical efficiency. We first present our solution for cluster environments, and then customize it for FL scenarios.

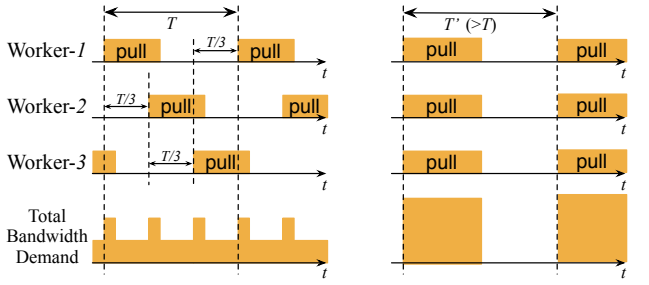
A. R^{2}SP in Cluster Environments

The R^{2}SP scheme employs two control mechanisms that respectively control the *temporal gap* between consecutive updates and the *update order*. Specifically, to minimize network contention, the R^{2}SP 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^{2}SP coordinates workers to make gradient updates in a *fixed round-robin order*. We next elaborate the two control mechanisms in detail.

1) Control 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.

[3]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.

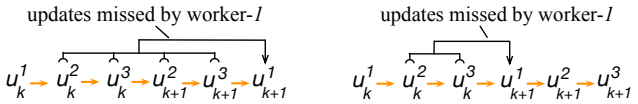
2) Control Update Order for Minimum Staleness: While enforcing the above inter-update gap can mitigate the communication bottleneck, the resultant asynchrony poisons statistical efficiency. Next, we further show that, by controlling the worker update *order*, R^{2}SP can minimize the negative effect 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 [16], [18], [17]. In particular, the *more* updates a worker misses, the *more poisonous* its update is, and in general the *lower* the statistical efficiency of the training process [16], [18]. 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 [16]. 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



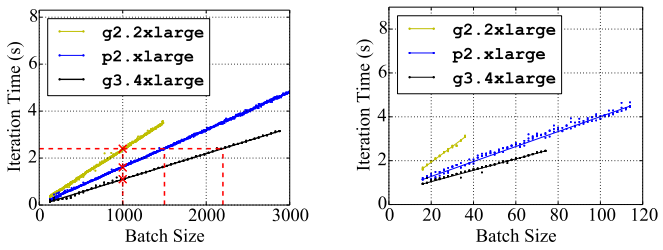
(a) SSP (Max Progress Gap: 1) (b) R^{2}SP

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.

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 SSP.

3) Batch Size Tuning in Heterogeneous Clusters: 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 [8], [9], [10], [11], it is not uncommon that many DL models were trained in *heterogeneous* clusters with different generations of GPUs [18], [19], [48]. 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. 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 [9], [10] 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. IV-A1) is determined by the slowest worker (i.e., the one with the longest T). Consequently, fast workers have to wait for their turn to come as they finish iterations earlier, hence wasting their computing cycles and impairing the learning efficiency.

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



(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.

work out how to tune its batch size to keep it busy during training without delaying others.

Moreover, when incorporating batch size tuning into R^{2}SP, we adopt a *linear scaling rule* on the learning rate, similar to [14]. 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-A3.

4) Convergence Analysis: We then analyze the convergence properties of R^{2}SP. We note that given the smoothness and bounded-gradient assumptions, the convergence validity of R^{2}SP can be theoretically guaranteed for general non-convex models. Due to the space limitation, we only state the assumptions, lemma and theorem here; please refer to our appendix for the detailed proofs of the lemmas and theorems.

Assumption 1. (Smoothness) *The loss function $F(x)$ is β-Lipschitz smooth, i.e., $\|\nabla F(x) - \nabla F(y)\| \leq \beta \|x - y\|$.*

Assumption 2. (Bounded Gradient) *The model gradients are bounded as $\|\nabla F(x)\|^2 \leq \sigma^2$ where σ is a constant.*

Given these assumptions, we can get the following lemma:

Lemma 1. *For any iteration $k \geq 1$, under Assumption 1 and Assumption 2 it holds that*

$$\|\tilde{x}_k - x_k\| \leq \sum_{k=1}^{T} \frac{\eta_k \eta_{k-1} \beta \sigma}{2}. \tag{1}$$

We can further derive the following theorem on the convergence validity of R^{2}SP:

Theorem 1. (Convergence Property) *Under Assumption 1 and Assumption 2, if one chooses a learning rate schedule such that for any iteration $t > 0$:*

$$\sum_{k=0}^{t-1} \frac{\eta_k^2}{\sqrt{\eta_t}} \leq D, \tag{2}$$

for some constant $D > 0$, then after running T iterations in R^{2}SP, we have:

$$\frac{1}{\sum_{k=1}^{T} \eta_k} \sum_{k=1}^{T} \eta_k \mathbb{E}[||\nabla F(x_k)||^2] \leq \frac{4(F(x_0) - F(x^*))}{\sum_{k=1}^{T} \eta_k} + \frac{(\beta^4 \sigma^2 D^2 + 2\sigma^2 \beta) \sum_{k=1}^{T} \eta_k^2}{\sum_{k=1}^{T} \eta_k}. \tag{3}$$

The above theorem implies that model training under R^{2}SP can converge when η_k satisfies:

$$\lim_{T \to \infty} \sum_{k=1}^{T} \eta_k = \infty \quad \text{and} \quad \lim_{T \to \infty} \frac{\sum_{k=1}^{T} \eta_k^2}{\sum_{k=1}^{T} \eta_k} = 0. \tag{4}$$

Therefore, we can set, for example, $\eta_k = \mathcal{O}(k^{-\frac{3}{4}})$ which satisfies the conditions in Eq. 4, and the model convergence can be guaranteed under R^{2}SP. We also analyze in our appendix why R^{2}SP can theoretically attain a better convergence performance than SSP.

B. Applying R^{2}SP in Federated Learning Scenarios

In recent years, Federated Learning (FL) is booming as an effective paradigm that allows clients to jointly train a model with data privacy preserved. In this section, we extend the idea of R^{2}SP to FL scenarios, and propose *Federated Learning Round-Robin Synchronous Parallel* (FL-R^{2}SP).

1) Challenges to Apply R^{2}SP in FL: Given the communication bottlenecks at the FL server, it is a natural idea to apply R^{2}SP for FL. Yet, we find that R^{2}SP is however not directly applicable due to the distinct characteristics of FL.

First, the behaviors of FL clients are highly unstable. In cluster environment, GPU workers' performance is quite stable, and that is why R^{2}SP—by strictly dictating ①which worker to pull the model next and ②at what time—can evenly interleave the workers' network transmissions. In contrast, in FL environment, a client's computing capability may vary drastically within a round, and it may even drop offline for reasons like power outage or network failure. In that case, if we force the FL server to wait for a designated client to make the next update as in R^{2}SP, the FL process may be severely slowed down (if the expected client returns its update very late) or even halt (if that client unfortunately loses connection). Therefore, our R^{2}SP-style solution for FL must be robust to such client dynamicity.

Second, the data distributions on different FL clients might be highly heterogeneous. In cluster environment, the entire dataset is well shuffled before being partitioned to each worker, and in this way all the workers can be treated indiscriminately in aggregation. However, in FL scenarios, the data samples are private to each client; generated under heterogeneous user preferences or environments, the data samples on different FL clients are often non-IID [49], [22], [23]. In particular, the local datasets of some clients may be highly skewed compared with others, and their updates would be poisonous for global model convergence [42], [50], [51]. Therefore, even when those data outliers possess good system capability, we shall identify and

This article has been accepted for publication in IEEE Transactions on Networking. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TON.2025.3644644

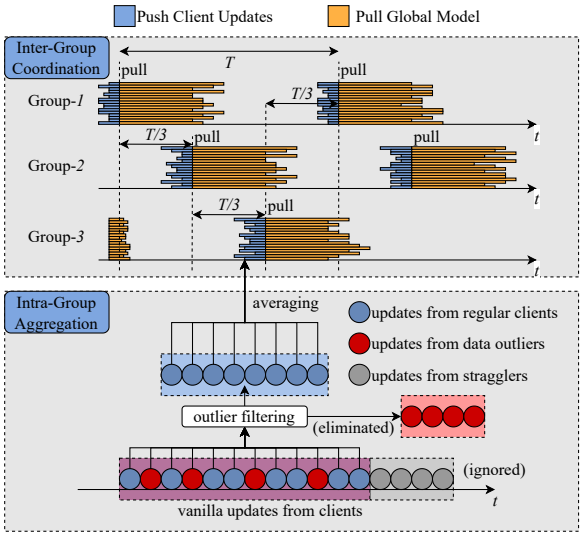


Fig. 6: An illustrative figure of FL-R^{2}SP with 3 groups. FL-R^{2}SP is composed of two control levels: synchronous aggregation within each group, and R^{2}SP coordination across different groups. In particular, to tackle system instability and data heterogeneity, we avoid aggregating the updates from stragglers and data outliers in each group.

eliminate them from our round-robin coordination for high FL accuracy.

In a nutshell, when mitigating the server-side communication bottleneck of FL, we need to simultaneously address the *system instability* and *statistical heterogeneity* of FL clients. To that end, we customize the previous R^{2}SP design for FL and propose FL-R^{2}SP, which will be elaborated next.

2) FL-R^{2}SP Design: The key innovation in FL-R^{2}SP design is the concept of client *group*, which provisions a stable *handle* for communication interleaving when facing unstable and heterogeneous FL clients. In this subsection, we first show how to make communication coordination *across different groups*, and then show how to handle system instability as well as statistical heterogeneity *within each group*.

Group: An Abstraction for Synchronization Interleaving. As elaborated in Sec. IV-B1, due to strong dynamicity in client participation, it is impossible to direct coordinate each client as in R^{2}SP. To decouple bandwidth multiplexing functionality with the burden of tackling system and statistical instability, we propose to make synchronization coordination at a coarser granularity—manipulating *client subsets* instead of individual clients; such a client subset is called a *group*[4]. As shown in Fig. 6, in FL-R^{2}SP we divide all the clients evenly into multiple groups: once a client joins the FL process, it is assigned to a group with a static mod-mapping function (i.e., the remainder of dividing client id by total group number).

Regarding synchronization coordination, the client updates within a group are aggregated in a *synchronous* manner, and different groups refine the global model (and push the latest model back to each clients) in a *round-robin* manner. That is, like in R^{2}SP, we fix the update order of different groups and also equalize their inter-update gap. Suppose there are M groups ($\{G_0, G_1, ..., G_{M-1}\}$), and the average round time of all the groups is T. Then, after group G_i aggregates its inner gradients to update the global model, FL-R^{2}SP would dictate $G_{(i+1)\%M}$ to make the next aggregation after a time gap of at least $\frac{T}{M}$—irrespective of the client dynamicity or heterogeneity within each group. In this way, we can evenly interleave the client-server transmission across different groups, effectively mitigating the server-side communication bottleneck.

Handle System Instability within Each Group. While client group provides an ideal abstraction to make round-robin coordination, we still need to tackle dynamic client participation within each group: if each group waits for the updates from all its clients, then the round time would be dominated by the slowest clients—even becoming infinite if a client loses connection during training. Note that this is essentially a straggler problem as appeared in vanilla FL [22], [23]; we therefore adopt a similar *fractional aggregation* method at intra-group level: to make an aggregation, each group only waits for a fraction (C) of the client updates that are returned the earliest—and the time to collect those early-bird updates (instead of all the updates) is deemed as the round time. When the straggling clients later return their updates after aggregation, their updates are ignored and they are directly replied with the latest model parameters. In this way, we can tackle the straggler problem within each group.

Handle Statistical Heterogeneity within Each Group. As elaborated in Sec. IV-B1, in FL scenarios, the local datasets on some clients may be highly skewed; incorporating their updates would compromise the global model convergence. We therefore need to identify those data outliers and avoid aggregating their updates within each group. Since client privacy is a key concern for FL, outlier identification would be better conducted directly with the client updates: client updates are readily collected on the FL server for aggregation and is thus a *free-lunch* for outlier analysis; in contrast, collecting other information (like the local feature-map [52] or loss value [53] on each client) would incur additional network cost and also increase the risk of privacy leakage.

In our FL-R^{2}SP design, we identify data outliers based on the cosine similarity between a client's local update and the global update. Since the data distributions of outliers are significantly different from others (yielding different local loss functions with heterogeneous optimums), the updates of outliers are often inconsistent with the majority and exhibit low cosine similarity with the global update [42]. Therefore, we can judge a client as data outlier if the cosine similarity between its local update and the global update keeps below a threshold (β) for a number of consecutive rounds. As seen in later evaluations, such an outlier-detection method[5] is efficient and also effective, being an indispensable component of FL-R^{2}SP to attain high model accuracy.

[4]Note that a group here is only a virtual relationship for client organization; there is no dedicated (edge) server to manage each group.

[5]That said, we can also use other outlier detection or client selection methods in the literature [52], [53], [38]. Note that our primary contribution here is to improve the accuracy performance of FL-R^{2}SP by integrating outlier filtering in each group (any effective method is acceptable)—instead of designing a brand new outlier filtering method.

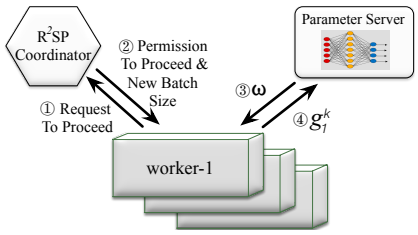


Fig. 7: Architecture and workflow of R²SP-Coordinator (in iteration k). Circled numbers represent execution order.

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

V. IMPLEMENTATION

A. R²SP Implementation

We have implemented R²SP as a ready-to-use Python library, called R²SP-Coordinator, for popular DL frameworks such as TensorFlow [9] and MXNet [10].

Workflow. Fig. 7 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 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. IV-A2) and the inter-update gap (Sec. IV-A1). 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-A4).

Overhead. In our implementation, we use Apache Thrift [54], 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 R²SP algorithm, the overhead of our implementation is negligible.

B. FL-R²SP Implementation

We have implemented FL-R²SP also atop PyTorch. To support flexible server-client interaction in FL (e.g., continue training even when some clients lose connection), we implement the communication between the FL server and the clients with RPyC (Remote Python Call) [55]—instead of with the built-in synchronization APIs of PyTorch (e.g., dist.all_reduce or DistributedDataParallel). RPyC is a transparent python library for remote interaction, with which each FL client interacts with the FL server independently. With such flexible server-client interaction, there is no need to maintain a dedicated R²SP-Coordinator as in cluster environment (Fig. 7); we instead integrate the coordination logic of FL-R²SP directly in the FL server. The workflow with FL-R²SP is described in Alg. 2.

Workflow of FL Clients. As shown in Alg. 2, once a FL client has refined its local model for a designated number of iterations, it would call the TryIntraGroupAggregate procedure to report the update to the FL_Server and wait for the latest model. This way, the key operations of FL-R²SP can be made transparent to the end clients.

Workflow of FL Server. On the server side, once receiving the update from a client, the FL server will map that client to a group, which maintains the fraction of updates collected so far (i.e., G.frac_collected). If that group has not collected

Algorithm 2 Workflow of FL-R^{2}SP

Require: $\tau, M, \vec{G}, C$ ▷ τ: the number of local iterations in each round; M: the number of groups; $\vec{G}$: the list of all groups; C: the fraction of updates required for intra-group aggregation.

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

procedure CLIENTITERATE(k)
  $k \leftarrow k + 1$
  $\omega_k^i \leftarrow \omega_{k-1}^i + u_k^i$ ▷ u_k^i: local update in iteration k on client i
  if k mod τ = 0 **then**
    $\omega_k^i \leftarrow$ `FL_Server`.TryIntraGroupAggregate(i, ω_k^i)

FL_Server:

procedure TRYINTRAGROUPAGGREGATE(i, ω_k^i)
  $G \leftarrow \vec{G}[i\%M]$ ▷ map i to a group
  G.frac_collected $\leftarrow$ G.frac_collected + M/N
  if G.frac_collected $< C$ **then** ▷ if updates are insufficient
    wait
  else
    if G.frac_collected = C **then** ▷ shall trigger aggregation
      $G.\bar{\omega} \leftarrow$ the average of all the ω_k^i collected by G
      $G.\bar{\omega} \leftarrow$ InterGroupAggregate(G) ▷ global update
      return $G.\bar{\omega}$ to all the pending clients
    else ▷ if c is a straggler
      return $G.\bar{\omega}$ to c ▷ Ignore update from stragglers
    identify and blacklist data outliers ▷ as in Sec. IV-B2

function INTERGROUPAGGREGATE(G)
  wait before G gets the *turn* to refine global model ▷ following requirements on inter-update order and inter-update gap
  $\omega \leftarrow \frac{M-1}{M} * \omega + \frac{1}{M} * G.\bar{\omega}$ ▷ update the global model ω
  refresh inter-update gap with the latest round time of G
  return ω

enough updates (i.e., the arrived portion is smaller than C), then the aggregation request from the client would be blocked; if that portion just reaches C, then intra-group aggregation is triggered, which further calls `InterGroupAggregate()` to refine the global model; if that portion becomes larger than C—which means that the update comes from a straggler, then the FL server would simply ignore the update and return the latest model to that client. Moreover, after intra-group aggregation, the FL server identifies data outliers by calculating the cosine similarity between the global update and the local update; data outliers would be moved to blacklist and eliminated from participating model training in the future.

In particular, the `InterGroupAggregate()` function coordinates different groups to make global model refinement in a evenly-gapped, round-robin manner. Once a group makes a global refinement, the next global model refinement can only be conducted by a designated group (i.e., with the next id) and after a designated time gap (i.e., T/K in Sec. IV-B2). To adapt to runtime dynamicity, we record the latest per-round time of each group and incrementally update the estimated per-round time T—with EMA method where the weight of latest observation is 0.1.

Next, we respectively evaluate the performance of R^{2}SP and FL-R^{2}SP. In Sec. VI-A, we focus on R^{2}SP performance evaluation in GPU clusters, demonstrating that it can speed up model convergence by over 30%. Further in Sec. VI-B, we show that FL-R^{2}SP can effectively mitigate the communication bottleneck of the FL server.

VI. EVALUATION

A. R^{2}SP Evaluation in Cluster Environment

1) Experimental Setup: We first evaluate R^{2}SP performance in cluster environments. We 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.

Hardware Platform. 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 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.[6]

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. We also use a 2080Ti GPU cluster later for training transformer-based models (Sec. VI-A5).

Datasets and Models. We trained two typical image classification datasets using TensorFlow [9]: (1) CIFAR-10 [39] and (2) ILSVRC12. The latter is a subset [40] of ImageNet22K containing 1.28 million of training images in 1000 categories. In particular, we trained ResNet32 [56], AlexNet [2] and VGG16 [57] models over the CIFAR-10 dataset, and trained Inception-v3 model [58] 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.

[6]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.

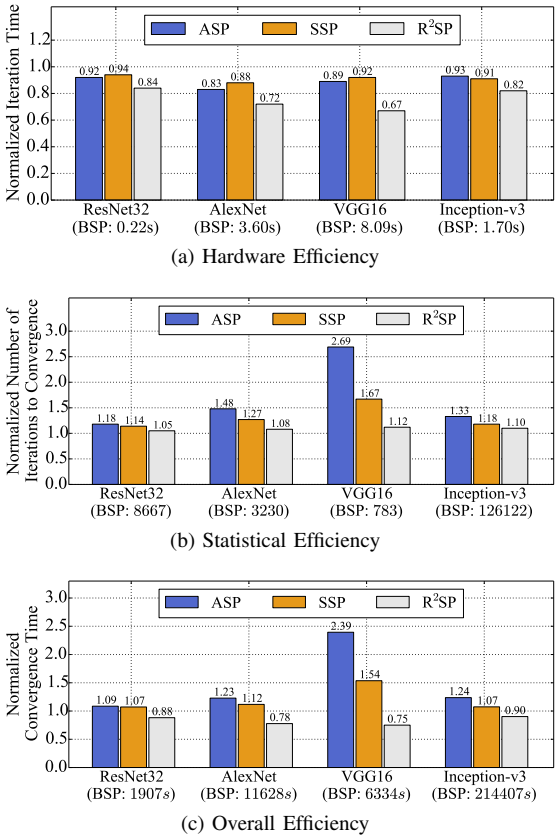


Fig. 8: Performance comparison among BSP, ASP, SSP and R^{2}SP. The presented values are normalized by BSP.

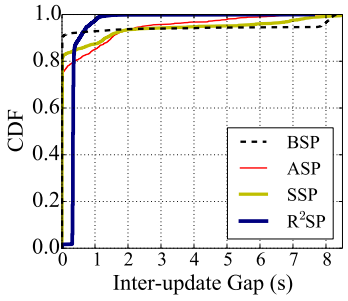


Fig. 9: CDF of the inter-update gap time when training VGG16.

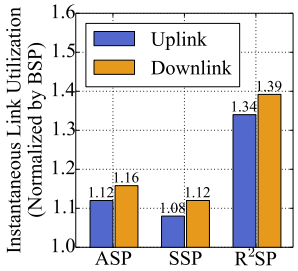


Fig. 10: Instantaneous PS bandwidth utilized by each worker during transferring.

2) R^{2}SP Performance in Homogeneous Cluster: We evaluate the efficiency of R^{2}SP against the three existing synchronization schemes in cluster-A. Fig. 8 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. 8a 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. 8b shows that compared with BSP, more iterations are needed for convergence 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. Fig. 8c shows the overall efficiency. Despite a slight statistical efficiency loss, R^{2}SP yields the highest convergence efficiency. 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 [12], [13], [14].

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. 9 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 measure 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. 10. 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. 8a.

3) R^{2}SP Performance 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 III, 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. 11 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, batch size tuning itself can accelerate convergence by around 40%.

4) Sensitivity Analysis: Finally, we perform sensitivity analysis on the factors that may affect the performance of R^{2}SP. We evaluate the behaviors of R^{2}SP with various link

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

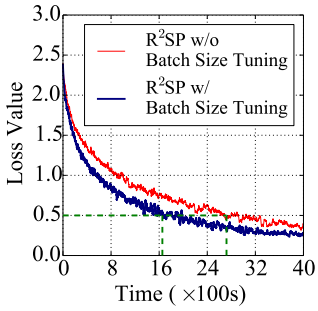


Fig. 11: Convergence curves of ResNet32 in Cluster-B.

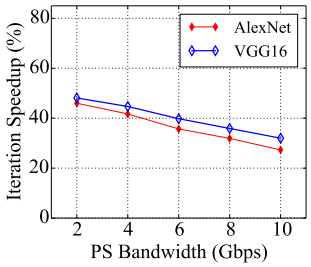


Fig. 12: The more severe the network bottleneck, the larger benefit (vs. BSP) from R^{2}SP.

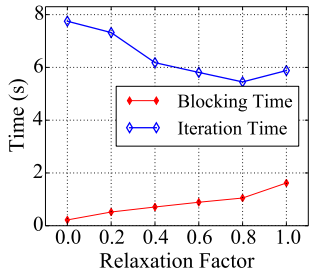


Fig. 13: Blocking time and iteration time under R^{2}SP with different relaxation factors .

bandwidths and values of the *relaxation factor* (Sec. V).

We first check the impact of 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. 12 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 [14] and geo-distributed machine learning [59].

Moreover, 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. 13 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. 13 that the sweet spot for the relaxation factor is achieved at 0.8, which leads to the shortest iteration.

5) R^{2}SP Performance for Transformer-based Models: Previously our evaluations are conducted on computer vision models. To demonstrate the effectiveness of R^{2}SP in modern NLP models, we further evaluate the performance of R^{2}SP when training transformer-based models. Specifically, we adopt two transformer-based models: T5-Small [60] and BERT [61]. T5-Small is trained on the wmt16 [62] (de-en) dataset; BERT is trained on the QNLI dataset from the GLUE [63] benchmark. The default batch size for all datasets is 400. In the training cluster, there are eight (2080Ti) GPU servers as the workers, as well as two (Xeon Gold 6230) CPU servers as the parameter servers. Those servers are interconnected with 100Gbps network. Moreover, to emulate the worker heterogeneity as explained in Sec. IV-A3, we manually inject per-iteration pre-updating delay on four workers: for two workers we inject a 0.5s delay and for two another workers we inject a 1s delay. Besides, regarding the baselines, apart from the three classical baselines (BSP, ASP, SSP), we additionally include another baseline, SAFA [45], for performance comparison. SAFA is a semi-synchronous coordination scheme: apart from coordinating a set of high-quality participants in a synchronous manner, it also properly absorbs the updates from straggling participants as long as their staleness level is modest.

Fig. 14 shows the hardware, statistical and overall efficiency when training the two NLP models under different schemes. According to Fig. 14a, with batch size tuning, R^{2}SP can attain the best per-iteration time; for the other schemes, there is a clear trade-off between the synchronization quality and the synchronization efficiency. Meanwhile, according to Fig. 14b, R^{2}SP suffers a slight statistical efficiency degradation when compared to BSP and SAFA, which is indeed reasonable and consistent with Fig. 8. Regarding the end-to-end convergence efficiency, Fig. 14c clearly suggests that R^{2}SP can substantially outperform the other baselines, with a 16.9% and 18.8% improvement to the second best. This confirm that, apart from computer vision models, R^{2}SP can also attain remarkable performance improvement for transformer-based large language models.

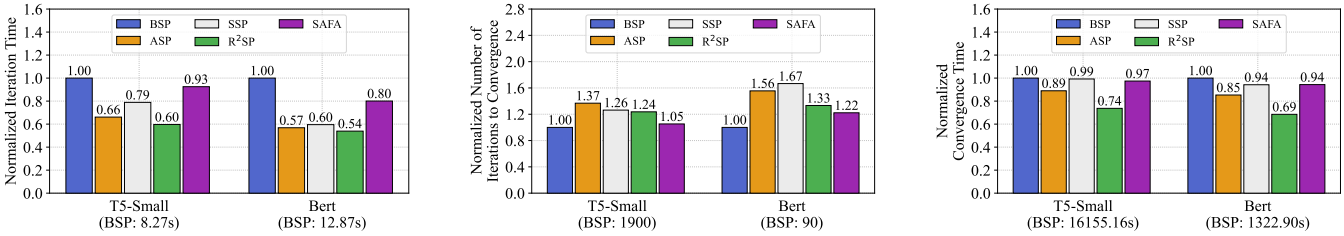
B. FL-R^{2}SP Evaluation in Emulated FL Scenarios

In this section, we evaluate FL-R^{2}SP performance in an emulated federated learning setup.

1) Experimental Setup: We first elaborate our experimental setup to evaluate FL-R^{2}SP.

Hardware Platform. We evaluate FL-R^{2}SP in an EC2 cluster comprising 129 instances: 1 `m5.4xlarge` instance (with 16 vCPUs, 64GiB memory and 10Gbps network bandwidth) as the FL server, and 128 `t2.small` instances (each with 1 vCPU, 2GiB memory and less than 100Mbps network bandwidth) as the low-end FL clients.

Moreover, a FL client usually exhibits drastic resource fluctuation and network jitters; to emulate such participation dynamicity, we manually enforce each client to wait for a random period before returning their updates. The delay we inject on each client follows a log-normal distribution whose probability density function is $f(x;\mu,\sigma) = \frac{1}{x\sigma\sqrt{2\pi}}\exp\left(-\frac{(\ln(x)-\mu)^2}{2\sigma^2}\right)$, where $\mu = -2, \sigma = 1$ (meaning that the average delay is $e^{(\mu+\sigma^2/2)} = 0.22$s). With such configurations, we make each `t2.small` instance behave like a smart phone.



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

Fig. 14: Performance comparison among BSP, ASP, SSP, R^{2}SP and SAFA. The presented values are normalized by BSP.

Datasets and Models. Compared with cluster environments, the models and datasets in federated learning scenarios are usually much smaller. Datasets in our experiments are CIFAR-10 [39] and the KeyWord Spotting dataset (KWS)—a subset of the Speech Commands dataset [64] with 10 keywords. To synthesize non-IID data distribution[7] (with outliers), we independently draw each client's training samples following the Dirichlet distribution [65], which controls local class evenness via a concentration parameter α ($\alpha \to \infty$ means IID data distribution). In our experiments, for 75% clients we set $\alpha = 10$ (under which each of the 10 data classes takes up no less than 7% of the entire dataset); for the remaining 25% clients we set $\alpha = 0.1$ (under which there is a single class taking up over 97% of the local dataset), and they play as data *outliers*. Such an IID/non-IID ratio aligns with the realistic composition revealed by the FedScale benchmark [66], which we will elaborate in greater detail in Appendix. With the CIFAR-10 dataset we train a CNN model and the ResNet20 model [67], and with the KWS dataset we train a LSTM model. Meanwhile, we use the SGD optimizer for all the models, and the learning rates are all set to 0.01.

Client Coordination. When enforcing our FL-R^{2}SP scheme, we set the group size to 16, meaning that the 128 clients are evenly mapped to 8 groups. Within each group, we trigger model aggregation once 75% of its client members have returned their updates (i.e., $C = 0.75$ in Alg. 2), and we identify data outliers if the cosine similarity between its local update and the global one is below 0.5 for 3 consecutive rounds. While we set C to 75% in our main experiments, we do have checked the impact of C on overall system performance, and the detailed results are placed in Appendix. Meanwhile, the number of local iterations within each round is set to 100 (i.e., $\tau = 100$ in Alg. 2).

2) Accuracy Performance: In this part, we compare the accuracy performance of FL-R^{2}SP against other typical client coordination schemes in FL—BSP (also with a selection rate $C = 0.75$) and ASP. Moreover, recall that in Sec. IV-B2 we have proposed an simple yet effective algorithm to identify and eliminate data outliers (which would impair the model convergence quality). To verify the effectiveness of such method, we also incorporate a naive version of FL-R^{2}SP—with the *outlier filtering* functionality disabled—as a baseline for ablation study. Meanwhile, for fair comparison, we integrate the outlier filtering functionality into BSP and also include it as an additional baseline.

[7]In this paper the induced non-IID behavior corresponds to label skewness. Therefore we set data non-IID level by controlling client label composition.

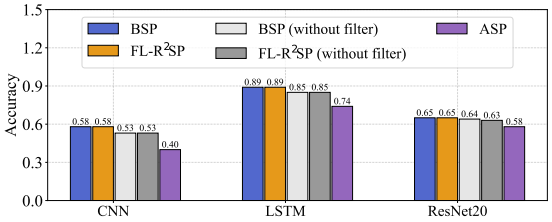


Fig. 15: Model convergence accuracy under different schemes.

TABLE IV: Model training time under FL-R^{2}SP and BSP.

Model	Scheme	Average Per-Round Time	Convergence Time	Improve-ment
CNN	BSP	10.69 s	4098.81s	22.4%
	FL-R^{2}SP	8.31 s	3181.55 s	
LSTM	BSP	9.41 s	2695.38 s	13.9%
	FL-R^{2}SP	8.17 s	2319.89 s	
ResNet20	BSP	44.90 s	9172.85 s	11.1%
	FL-R^{2}SP	36.75 s	8158.34 s	

In Fig. 15, we compare the model convergence accuracy under different client coordination schemes in FL. Here the FL process is deemed to have *converged* if the testing accuracy has stagnated for 10 consecutive rounds. As shown in Fig. 15, our FL-R^{2}SP scheme achieves comparable accuracy performance with BSP for each model. Meanwhile, we can verify that outlier filtering is indeed quite necessary to ensure model convergence quality: without outlier filtering, there would be an accuracy loss by around 10% for both BSP and FL-R^{2}SP. This conclusion also holds in our additional experiments on the FedScale benchmark, which is detailed in Appendix. Besides, we can notice that ASP suffers severe accuracy degradation due to the problem of stale updates—a fatal deficiency for federated model training. Therefore, in the remaining part of our evaluation, we rule out ASP and focus on the comparison between BSP and FL-R^{2}SP.

3) Efficiency Performance: Recall that a primary objective of FL-R^{2}SP is to speed up model convergence by mitigating the server-side communication bottleneck. To verify the training speedup effect of FL-R^{2}SP, we measure the *model convergence time* and the *average per-round time* when train-

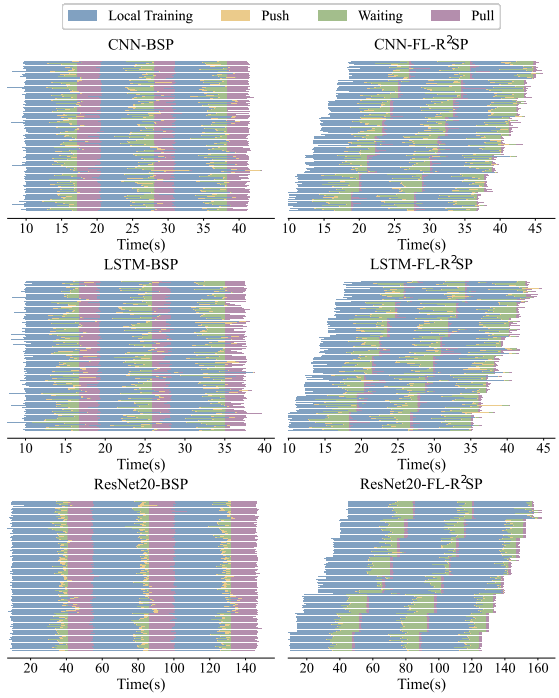


Fig. 16: Gantt charts depicting the instantaneous client status during a model training period under BSP and FL-R^{2}SP.

TABLE V: Average per-round communication time among all the clients under BSP and FL-R^{2}SP.

Model	Scheme	Average Push Time	Average Pull Time	Overall Communication Speedup
CNN	BSP	0.38 s	2.73 s	76.5%
	FL-R^{2}SP	0.34 s	0.38 s	
LSTM	BSP	0.36 s	2.12 s	71.7%
	FL-R^{2}SP	0.34 s	0.36 s	
ResNet20	BSP	1.19 s	13.28 s	86.3%
	FL-R^{2}SP	0.47 s	1.50 s	

ing the three models respectively under BSP and FL-R^{2}SP. As shown in Table IV, FL-R^{2}SP can attain remarkable speedup in each case, with an end-to-end performance improvement of up to 22.4%.

To further understand the effectiveness of FL-R^{2}SP, we make fine-grained measurements on the instantaneous client status during the FL process. We note that each client experiences four phases in a round: (1) local training, (2) push update to the FL server, (3) wait for model aggregation, and (4) pull the latest model from the FL server; in Fig. 16, we draw the Gantt charts depicting the phases on all the clients (totally 128 clients in 8 groups) during a randomly selected period (round 100 to 102). From Fig. 16, we can learn that the clients in different groups indeed make aggregation in an evenly-gapped, round-robin manner. Moreover, we can observe that the purple region (representing the *pulling* phase) under FL-R^{2}SP is much smaller than that under BSP, indicating that the network contention with FL-R^{2}SP is much slighter than BSP.

4) Communication Speedup Deep Dive: In this part, we dive deep on the performance benefit of FL-R^{2}SP in the perspective of communication speedup. In Table V, we measure the average time spent by each client on network transmission (i.e., on pushing the updates and pulling the latest models), and further calculate the communication speedup of FL-R^{2}SP over BSP. As suggested by Table V, the server-side network bottleneck is more severe at pulling time—under BSP, the average model pulling time can be 10× longer than the update pushing time: this is because update pushing is triggered *independently* by each client, yet model pulling is launched *simultaneously* by the FL server for all the clients. With FL-R^{2}SP, we can effectively mitigate such *outcast*-style network contentions, making the pulling time comparable to pushing time. Consequently, FL-R^{2}SP remarkably speeds up communication—with a time reduction of over 70%.

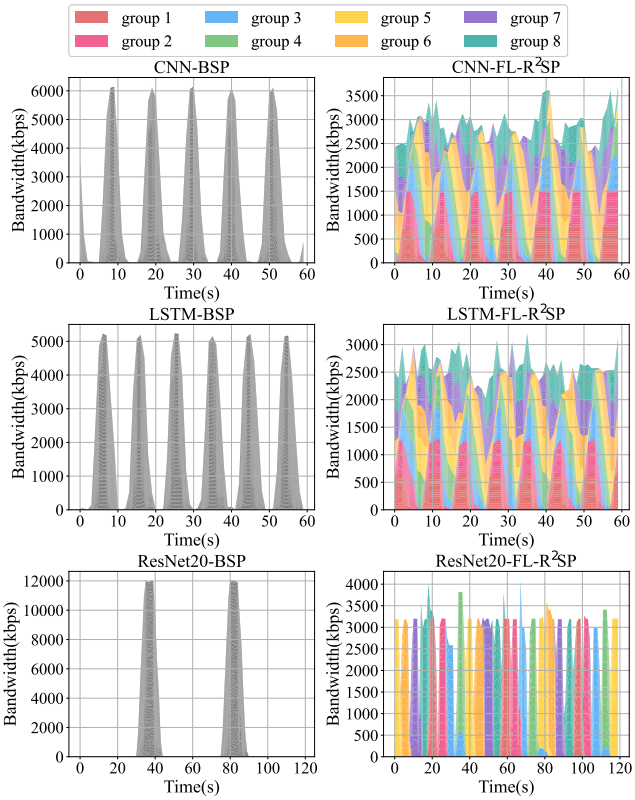


Fig. 17: Server uplink usage under FL-R^{2}SP and BSP.

In Fig. 17, we measure the FL server's uplink bandwidth usage (i.e., used for clients to pull the latest model) when training the three models. The bandwidth usages are traced with the nethogs [68] tool. Under the BSP scheme, all the clients pull the model parameters at almost the same pace, thus incurring high peak network consumption; under the FL-R^{2}SP scheme, the network consumption of clients in different groups can be well staggered with each other, reducing the peak bandwidth consumption and in the meantime increasing the network utilization. For example, for ResNet-20, the peak bandwidth consumption of BSP is around 3× of that under FL-R^{2}SP. Such microscopic observations align with the transmission speedup in Table V and the end-to-end training speedup in Table IV.

5) Sensitivity Analysis on Group Size: Recall that a key innovation of FL-R^{2}SP is to make synchronization coordina-

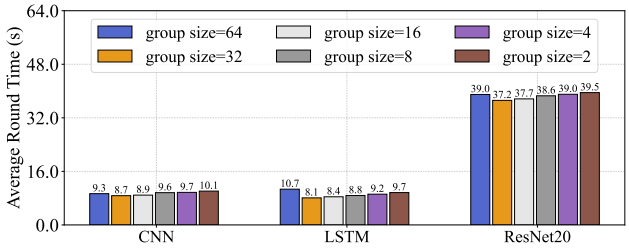


Fig. 18: Average round time comparison between FL-R^{2}SP with different group size.

tion at the granularity of *client groups* instead of individual clients. Obviously, the number of clients within a group (i.e., the *group size*) may remarkably influence the performance of FL-R^{2}SP. As two extreme cases, when the group size is set to 1, FL-R^{2}SP would degenerate to vanilla R^{2}SP; when the group size is set equal to the total client number, FL-R^{2}SP would degenerate to BSP. The client group size we set in our previous experiments is 16, and in this part we explore how FL-R^{2}SP would perform against different group size values.

In Fig. 18, given the 128 clients, we respectively set the group size to different values (from 2 to 64) and measure the average round time for the three models under FL-R^{2}SP. We can observe from Fig. 18 that, the correlation pattern between round time and group size is similar across different models. On the one hand, with a very large group size (thus with a small number of groups), the staggering effect of inter-group communication is quite restricted, and meanwhile the network contention for clients within a group may become a bottleneck; on the other hand, with a very small group size (thus with a large number of groups), due to the client instability, the additional time overhead to enforce the targeted inter-update gap would increase. As in Fig. 18, when the group size changes from 2 to 64, the round time would first decrease and then increase. Therefore, the group size shall not be set too small or too large—Fig. 18 suggests that 32 is the most appropriate group size value.

VII. Conclusion

In this work, we respectively designed R^{2}SP and FL-R^{2}SP to mitigate the server-side communication bottlenecks for both cluster and FL environments. R^{2}SP works by evenly interleaving the worker-PS communications in a round-robin manner, and for FL scenarios, FL-R^{2}SP makes round-robin coordination at the granularity of groups instead of individual clients. Extensive evaluations have been conducted to demonstrate the effectiveness of both R^{2}SP and FL-R^{2}SP.

Acknowledgement

The research was sponsored in part by the National Natural Science Fundation of China (NSFC) (62432008, 62202300). It is also supported by RGC RIF grant R6021-20, an RGC TRS grant T43-513/23N-2, RGC CRF grants C7004-22G, C1029-22G and C6015-23G, NSFC/RGC grant CRS_HKUST601/24 and RGC GRF grants 16207922, 16207423 and 16203824. The preliminary version of this article was published in IEEE INFOCOM [69]. We thank the anonymous reviewers for the constructive advices.

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.
[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *NIPS*, 2012.
[3] C. Farabet, C. Couprie, L. Najman, and Y. LeCun, "Learning hierarchical features for scene labeling," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 35, no. 8, pp. 1915–1929, 2013.
[4] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, "Natural language processing (almost) from scratch," *J. Mach. Learn. Res.*, vol. 12, no. Aug, pp. 2493–2537, 2011.
[5] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *NIPS*, 2014.
[6] S. Shi and X. Chu, "MG-WFBP: Efficient data communication for distributed synchronous sgd algorithms," in *IEEE INFOCOM*, 2019.
[7] Z. Tang, Y. Zhang, S. Shi, X. He, B. Han, and X. Chu, "Virtual Homogeneity Learning: Defending against Data Heterogeneity in Federated Learning," in *ICML*, 2022.
[8] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *USENIX OSDI*, 2014.
[9] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "TensorFlow: A system for large-scale machine learning." in *USENIX OSDI*, 2016.
[10] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv:1512.01274*, 2015.
[11] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," in *NIPS-W*, 2017.
[12] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing, "Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server," in *ACM Eurosys*, 2016.
[13] J. Chen, X. Pan, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous sgd," *arXiv preprint arXiv:1604.00981*, 2016.
[14] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch sgd: training imagenet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.
[15] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng, "Large Scale Distributed Deep Networks," in *NeurIPS*, 2011.
[16] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More effective distributed ml via a stale synchronous parallel parameter server," in *NIPS*, 2013.
[17] J. Langford, A. Smola, and M. Zinkevich, "Slow learners are fast," in *NIPS*, 2009.
[18] J. Jiang, B. Cui, C. Zhang, and L. Yu, "Heterogeneity-aware distributed parameter servers," in *ACM SIGMOD*, 2017.
[19] W. Zhang, S. Gupta, X. Lian, and J. Liu, "Staleness-aware async-SGD for distributed deep learning," *arXiv preprint arXiv:1511.05950*, 2015.
[20] C. Chen, Q. Weng, W. Wang, B. Li, and B. Li, "Semi-dynamic load balancing: Efficient distributed learning in non-dedicated environments," in *ACM SoCC*, 2020.
[21] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia, "Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads," in *OSDI*, 2020.
[22] J. Konečnỳ, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, "Federated learning: Strategies for improving communication efficiency," *arXiv preprint arXiv:1610.05492*, 2016.
[23] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Artificial intelligence and statistics*. PMLR, 2017, pp. 1273–1282.
[24] T. Li, A. K. Sahu, M. Zaheer, M. Sanjabi, A. Talwalkar, and V. Smith, "Federated optimization in heterogeneous networks," *Proceedings of Machine learning and systems*, vol. 2, pp. 429–450, 2020.
[25] S. P. Karimireddy, S. Kale, M. Mohri, S. Reddi, S. Stich, and A. T. Suresh, "Scaffold: Stochastic controlled averaging for federated learning," in *International conference on machine learning*. PMLR, 2020, pp. 5132–5143.
[26] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konečnỳ, S. Mazzocchi, B. McMahan *et al.*, "Towards federated learning at scale: System design," *Proceedings of machine learning and systems*, vol. 1, pp. 374–388, 2019.

[27] W.-T. Ho, S.-Y. Fang, T.-Y. Liu, and J.-J. Kuo, "Degree-aware in-network aggregation for federated learning with fog computing," in *IEEE Globecom Workshops (GC Wkshps)*, 2021.

[28] L. Liu, J. Zhang, S. Song, and K. B. Letaief, "Client-edge-cloud hierarchical federated learning," in *ICC 2020-2020 IEEE International Conference on Communications (ICC)*. IEEE, 2020, pp. 1–6.

[29] S. Luo, P. Fan, H. Xing, L. Luo, and H. Yu, "Eliminating communication bottlenecks in cross-device federated learning with in-network processing at the edge," in *IEEE ICC*, 2022.

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

[31] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Ng, "Large scale distributed deep networks," in *NIPS*, 2012.

[32] M. Li, T. Zhang, Y. Chen, and A. J. Smola, "Efficient mini-batch training for stochastic optimization," in *ACM KDD*, 2014.

[33] L. Zhang, S. Shi, X. Chu, W. Wang, B. Li, and C. Liu, "Dear: Accelerating distributed deep learning with fine-grained all-reduce pipelining."

[34] Z. Jiang, W. Wang, B. Li, and B. Li, "Pisces: efficient federated learning via guided asynchronous training," in *ACM SoCC*, 2022.

[35] D. Huba, J. Nguyen, K. Malik, R. Zhu, M. Rabbat, A. Yousefpour, C.-J. Wu, H. Zhan, P. Ustinov, H. Srinivas *et al.*, "Papaya: Practical, private, and scalable federated learning," *MLSys*, 2022.

[36] J. Nguyen, K. Malik, H. Zhan, A. Yousefpour, M. Rabbat, M. Malek, and D. Huba, "Federated Learning with Buffered Asynchronous Aggregation," in *AISTATS*, 2022.

[37] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth, "Practical secure aggregation for federated learning on user-held data," *arXiv preprint arXiv:1611.04482*, 2016.

[38] F. Lai, X. Zhu, H. V. Madhyastha, and M. Chowdhury, "Oort: Efficient Federated Learning via Guided Participant Selection," in *OSDI*, 2021.

[39] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," 2009.

[40] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *IEEE CVPR*, 2009.

[41] C. Chen, H. Xu, W. Wang, B. Li, B. Li, L. Chen, and G. Zhang, "Communication-efficient federated learning with adaptive parameter freezing," in *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2021, pp. 1–11.

[42] W. Luping, W. Wei, and L. Bo, "Cmfl: Mitigating communication overhead for federated learning," in *2019 IEEE 39th international conference on distributed computing systems (ICDCS)*. IEEE, 2019, pp. 954–964.

[43] D. A. E. Acar, Y. Zhao, R. M. Navarro, M. Mattina, P. N. Whatmough, and V. Saligrama, "Federated learning based on dynamic regularization," *arXiv preprint arXiv:2111.04263*, 2021.

[44] H. Cui, A. Tumanov, J. Wei, L. Xu, W. Dai, J. Haber-Kucharsky, Q. Ho, G. R. Ganger, P. B. Gibbons, G. A. Gibson *et al.*, "Exploiting iterativeness for parallel ml computations," in *ACM SoCC*, 2014.

[45] W. Wu, L. He, W. Lin, R. Mao, C. Maple, and S. Jarvis, "Safa: A semi-asynchronous protocol for fast federated learning with low overhead," *IEEE Transactions on Computers*, vol. 70, no. 5, pp. 655–668, 2021.

[46] "Cisco devices only support full-duplex." https://www.cisco.com/c/en/us/support/docs/lan-switching/ethernet/10561-3.html.

[47] D. Halperin, S. Kandula, J. Padhye, P. Bahl, and D. Wetherall, "Augmenting data center networks with multi-gigabit wireless links," in *ACM SIGCOMM*, 2011.

[48] F. Song and J. Dongarra, "A scalable framework for heterogeneous GPU-based clusters," in *ACM SPAA*, 2012.

[49] Y. Zhao, M. Li, L. Lai, N. Suda, D. Civin, and V. Chandra, "Federated learning with non-iid data," *arXiv preprint arXiv:1806.00582*, 2018.

[50] X. Li, K. Huang, W. Yang, S. Wang, and Z. Zhang, "On the convergence of fedavg on non-iid data," *arXiv preprint arXiv:1907.02189*, 2019.

[51] Q. Li, Y. Diao, Q. Chen, and B. He, "Federated learning on non-iid data silos: An experimental study," in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2022, pp. 965–978.

[52] B. Liu, Y. Guo, and X. Chen, "PFA: Privacy-preserving Federated Adaptation for Effective Model Personalization," in *ACM WWW*, 2021.

[53] Y. Mansour, M. Mohri, J. Ro, and A. T. Suresh, "Three approaches for personalization with applications to federated learning," *arXiv preprint arXiv:2002.10619*, 2020.

[54] "Apache Thrift," https://thrift.apache.org/.

[55] "RPyC," https://rpyc.readthedocs.io/en/latest/#.

[56] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *IEEE CVPR*, 2016.

[57] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[58] "TensorFlow Inception-V3," https://github.com/tensorflow/models/tree/master/research/inception.

[59] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G. R. Ganger, P. B. Gibbons, and O. Mutlu, "Gaia: Geo-distributed machine learning approaching LAN speeds." in *USENIX NSDI*, 2017.

[60] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, no. 1, Jan. 2020.

[61] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. [Online]. Available: https://aclanthology.org/N19-1423/

[62] "Acl 2016 first conference on machine translation (wmt16)," https://www.statmt.org/wmt16/index.html, 2016.

[63] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. Bowman, "GLUE: A multi-task benchmark and analysis platform for natural language understanding," in *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, T. Linzen, G. Chrupała, and A. Alishahi, Eds. Brussels, Belgium: Association for Computational Linguistics, Nov. 2018, pp. 353–355. [Online]. Available: https://aclanthology.org/W18-5446/

[64] P. Warden, "Speech commands: A dataset for limited-vocabulary speech recognition," *arXiv preprint arXiv:1804.03209*, 2018.

[65] M. Yurochkin, M. Agarwal, S. Ghosh, K. Greenewald, T. N. Hoang, and Y. Khazaeni, "Bayesian nonparametric federated learning of neural networks," in *Proc. ICML*, 2019.

[66] F. Lai, Y. Dai, S. S. Singapuram, J. Liu, X. Zhu, H. V. Madhyastha, and M. Chowdhury, "FedScale: Benchmarking model and system performance of federated learning at scale," in *International Conference on Machine Learning (ICML)*, 2022.

[67] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[68] "Nethogs," https://github.com/raboof/nethogs, 2022.

[69] C. Chen, W. Wang, and B. Li, "Round-robin synchronization: Mitigating communication bottlenecks in parameter servers," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 532–540.

[70] S. Wang, T. Tuor, K. K. Leung, C. Makaya, T. He, and K. Chan, "Adaptive federated learning in resource constrained edge computing systems," *Journal on Selected Areas in Communications*, vol. 37, no. 6, pp. 1205–1221, 2019.

[71] B. Luo, X. Li, S. Wang, J. Huang, and L. Tassiulas, "Cost-effective federated learning design," in *IEEE INFOCOM*, 2021.

[72] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns," in *Fifteenth annual conference of the international speech communication association*, 2014.

[73] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "Qsgd: Communication-efficient sgd via gradient quantization and encoding," *Advances in neural information processing systems*, vol. 30, 2017.

[74] Y. Gong, L. Liu, M. Yang, and L. Bourdev, "Compressing deep convolutional networks using vector quantization," *arXiv preprint arXiv:1412.6115*, 2014.

[75] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *NIPS*, 2015.

[76] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing, "Poseidon: An efficient communication architecture for distributed deep learning on gpu clusters," in *USENIX ATC*, 2017.

[77] L. Luo, P. West, A. Krishnamurthy, L. Ceze, and J. Nelson, "PLink: Discovering and Exploiting Datacenter Network Locality for Efficient Cloud-based Distributed Training," in *MLSys*, 2020.

[78] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo, "A generic communication scheduler for distributed dnn training acceleration," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019.

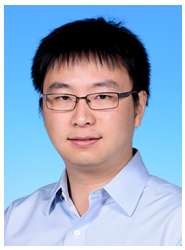
[79] A. Jayarajan, J. Wei, G. Gibson, A. Fedorova, and G. Pekhimenko, "Priority-based parameter propagation for distributed dnn training," *Proceedings of Machine Learning and Systems*, vol. 1, pp. 132–145, 2019.

[80] P. Jiang and G. Agrawal, "Adaptive periodic averaging: A practical approach to reducing communication in distributed learning," *arXiv preprint arXiv:2007.06134*, 2020.
[81] F. Haddadpour, M. M. Kamani, M. Mahdavi, and V. R. Cadambe, "Local sgd with periodic averaging: Tighter analysis and adaptive synchronization," in *NeurIPS*, 2019.
[82] A. Harlap, H. Cui, W. Dai, J. Wei, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Addressing the straggler problem for iterative convergent parallel ml," in *ACM SoCC*, 2016.
[83] C. Chen, Q. Weng, W. Wang, B. Li, and B. Li, "Semi-Dynamic Load Balancing: Efficient Distributed Learning in Non-Dedicated Environments," in *ACM SoCC*, 2020.
[84] E. Jeong, S. Oh, H. Kim, J. Park, M. Bennis, and S.-L. Kim, "Communication-efficient on-device machine learning: Federated distillation and augmentation under non-iid private data," *arXiv preprint arXiv:1811.11479*, 2018.
[85] C. Li, R. Li, H. Wang, Y. Li, P. Zhou, S. Guo, and K. Li, "Gradient scheduling with global momentum for non-iid data distributed asynchronous training," *arXiv preprint arXiv:1902.07848*, 2019.
[86] H. Chen, T. Zhu, T. Zhang, W. Zhou, and P. S. Yu, "Privacy and fairness in federated learning: On the perspective of tradeoff," *ACM Computing Surveys*, vol. 56, no. 2, pp. 1–37, 2023.

Jiayi Zhang received the B.Eng. degree from Shanghai Jiao Tong University. He is currently pursuing the M.S. degree in Computer Science at the University of Southern California. His research interests include distributed deep learning and networking.

Chen Chen is an Associate Professor in John Hopcroft Center for Computer Science, Shanghai Jiao Tong University. He received the B.Eng. degree from Tsinghua University in 2014, and the PhD degree in 2018 from Department of Computer Science and Engineering, Hong Kong University of Science and Technology. His recent research interests include distributed deep learning, big data systems and networking. He previously worked as a researcher at the Theory Lab, Huawei Hong Kong Research Center.

Zuo Gan received the master's degree of computer science in Shanghai Jiao Tong University. He received the B.Eng degree from Xi'an Jiaotong University in 2022. His research interest is to enhance the accuracy and efficiency performance of Federated Learning.



Wei Wang received his B.Eng. and M.Eng. degrees in Electrical Engineering from Shanghai Jiao Tong University in 2007 and 2010, respectively, and his Ph.D. in Electrical and Computer Engineering from the University of Toronto in 2015. He joined the Department of Computer Science and Engineering at the Hong Kong University of Science and Technology (HKUST) in 2015, where he is currently an Associate Professor. He also leads the HKUST Big Data Institute as Associate Director and holds additional leadership roles within HKUST's data and AI research initiatives. Dr. Wang's research interests encompass distributed and cloud systems, with a recent focus on distributed machine learning systems, serverless computing, and large-scale cluster management for AI clouds. He has published over 100 peer-reviewed papers in leading conferences and journals, accumulating more than 7,000 citations according to Google Scholar. His work has been recognized with Best Paper Awards at ACM EuroSys 2025 and ACM SoCC 2023, as well as Best Paper Runner-Up Awards at IEEE ICDCS 2021 and USENIX ICAC 2013. He currently serves as an Associate Editor of IEEE Transactions on Parallel and Distributed Systems. Previously, he was a Guest Editor for IEEE Transactions on Big Data and IEEE Network, and has been recognized as a Distinguished TPC Member of IEEE INFOCOM on four occasions.

Bo Li is a Chair Professor in Department of Computer Science and Engineering, the Director for Big Data Institute and HKUST-Alibaba Joint Lab on Big Data and Artificial Intelligence, Hong Kong University of Science and Technology. He held the Cheung Kong Chair Professor in Shanghai Jiao Tong University (2010-2016) and was the Chief Technical Advisor for ChinaCache Corp. (NASDAQ:CCIH), one of the world leading CDN service providers. He made pioneering contributions in multimedia communications and the Internet video broadcast, in particular the Coolstreaming system, which was credited as the world's first large-scale Peer-to-Peer live video streaming system. It attracted significant attention from industry with substantial investment, and also from academia in receiving the Test-of-Time Best Paper Award from IEEE INFOCOM (2015). He received 10 Best Paper Awards from IEEE including IEEE ICDCS (2025), IEEE Transactions on Cloud Computing (2023) and IEEE INFOCOM (2021). He has been an editor or a guest editor for over a two dozen of IEEE and ACM journals and magazines. He was the Co-TPC Chair for IEEE INFOCOM (2004). He is a Fellow of IEEE. He received his PhD in the Electrical and Computer Engineering from University of Massachusetts at Amherst, and his B. Eng. (summa cum laude) in the Computer Science Department from Tsinghua University, Beijing, China.

Minyi Guo is currently the Zhiyuan Chair professor of the Department of Computer Science and Engineering, Shanghai Jiao Tong University (SJTU), China. Before joining SJTU, Dr. Guo had been a professor of The University of Aizu, Japan. His present research interests include parallel/distributed computing, compiler optimizations, emerging computer architectures, and cloud computing. He has more than 500 publications in major journals and international conferences in these areas. He received 7 best paper awards from international conferences. He is now the Editor-in-Chief of IEEE Transactions on Sustainable Computing, and on the editorial board of IEEE Transactions on Cloud Computing and IEEE Transactions on Computers. Dr. Guo is a Fellow of IEEE and a Fellow of CCF.