FluentPS: A Parameter Server Design with Low-frequency Synchronization for Distributed Deep Learning

Xin Yao, Xueyu Wu and Cho-Li Wang
Department of Computer Science
The University of Hong Kong
Hong Kong, China
Email: {xyao, xywu, clwang}@cs.hku.hk

***Abstract*—With pursuing high accuracy on big datasets, current research prefers designing complex neural networks, which need to maximize data parallelism for short training time. Many distributed deep learning systems, such as MXNet and Petuum, widely use parameter server framework with relaxed synchronization models. Although these models could cost less on each synchronization, its frequency is still high among many workers, e.g., the soft barrier introduced by Stale Synchronous Parallel (SSP) model. In this paper, we introduce our parameter server design, namely FluentPS, which can reduce frequent synchronization and optimize communication overhead in a large-scale cluster. Different from using a single scheduler to manage all parameters' synchronization in some previous designs, our system allows each server to independently adjust schemes for synchronizing its parameter shard and overlaps the push and pull processes of different servers. We also explore two methods to improve the SSP model: (1) lazy execution of buffered pull requests to reduce the synchronization frequency and (2) a probability-based strategy to pause the fast worker at a probability under SSP condition, which avoids unnecessary waiting of fast workers. We evaluate ResNet-56 with the same large batch size at different cluster scales. While guaranteeing robust convergence, FluentPS gains up to 6× speedup and reduce 93.7% communication time costs than PS-Lite. The raw SSP model causes up to 131× delayed pull requests than our improved synchronization model, which can provide fine-tuned staleness controls and achieve higher accuracy.**

***Index Terms*—Parameter server, data parallelism, staleness synchronization parallel model**

I. Introduction

The rapid evolution of algorithmic advances (deeper neural networks like ResNet [1] etc.), abundance of data (e.g., 130 million images in Mapillary Vistas Dataset [2]), and GPU cluster can support more accurate AI products, but also require new designs to understand this trend and meet some challenges. The famous parameter server (PS) abstraction [3–5] enables to run deep learning tasks with different parallel training algorithms, such as model parallelism [6] (STRADS [7]), data parallelism (TensorFlow [8], MXNet [4, 9], Caffe [10]), and the hybrid parallelism mechanism (Angel [11], Petuum [12, 13], Project Adam [14]). While most systems increase the batch size to train models with few iterations [15], it becomes a common practice to use data parallelism. After calculating gradients on a partition of the training dataset, each worker pushes gradients to update the global parameters on servers and pulls the updated parameters from the servers that means to acquire the gradients updated by other workers. The traditional PS system designs [4, 9] deploy one centralized scheduler to record the progress of each worker and manage the parameters synchronization. Even in a load-balanced cluster, some worker nodes are randomly slower than other nodes [14]. A barrier is usually needed to synchronize their updates at the end of each iteration. Therefore, the scheduler often executes data-parallel algorithms in Bulk Synchronous Parallel (BSP) [16] model, which is prone to the straggler problem [17]. This problem causes serious concerns when the system is at a massive scale.

Significant efforts [18–20] on optimizing synchronization have been made to address the straggler problem in large-scale deep learning tasks, regarding learning rate schedules [21–23] and adaptive staleness [24, 25]. Asynchronous Parallel (ASP) [18] removed all barriers among workers but needed more iterations to achieve the convergent results. Staleness Synchronous Parallel (SSP) [20, 26, 27] allowed delayed update and the fastest worker could not exceed the slowest one more than a predefined staleness threshold **s**. In SSP model, the *soft barrier* [20] appeared once a worker is not within the specified range (i.e., the staleness threshold *s*) of the current iteration, which weakened the full barrier in BSP. However, the soft barrier still appeared frequently and became the major limitation to hurt the performance. Furthermore, to implement SSP, previous designs (e.g., SSPtable [13] and PrograssTracker [28]) recorded the progress of every worker and kept a consistent view of staleness between workers and servers. With an increasing number of workers, the staleness information maintenance of these old designs could cause poor scalability and convenience loss. Chen et al. [19] proposed to use additional backup workers and dropped the stragglers in each iteration. Adam [21] used parameter-specific learning rates, which computed individual adaptive learning rate for every parameter. Dynamic Synchronous Parallel Strategy (DSPS) [25] adjusted the staleness threshold *s* at runtime.

Our paper presents FluentPS, a parameter server design focused on reducing the synchronization frequency to achieve

fast convergence in large-scale deep learning systems. We define conditions for controlling synchronization at the server nodes to handle pull and push operations from workers. The pull condition requires the server to delay the responses to the pull requests from fast workers, which avoids intolerable errors of returning stale parameters to achieve robust convergence and high accuracy. When the push operations are executed, it allows the server to conditionally (i.e., the push condition) respond the buffered pull requests. We consider synchronization reduction from two aspects: (1) lazy pull execution: we delay the pull requests of some fast workers and provide them with the updated parameters for fast and robust convergence, and lazy pull execution could achieve 1.24× speedup than soft barrier with the same synchronization model; (2) probabilistic SSP model: we relax the pull condition in the SSP model and the server is not necessarily but at a probability to block fast workers and waits for the stragglers to catch up. SSP model guarantees a bounded delay among workers, which sometimes produces unnecessary waiting or misses significant gradients. The dynamic probability adjustment in our algorithm is sensitive to the staleness gap and the gradient significance, e.g., it can scale up the probability of synchronization corresponding to the growing staleness k and the staleness threshold s: $P(k) = 1/(1+e^{(s-k)})$. Compared with the SSP model under the same s, our experiments of ResNet-56 on CIFAR-10 dataset show 1.38× speedup.

To the best of our knowledge, FluentPS is also the first attempt of unifying implementation of various synchronization models by only specifying the pull/push conditions. At runtime, our condition-aware methodology can adjust synchronization models to improve flexibility. The main contributions are summarized as follows:

- We offload the synchronization control from a single scheduler on to every parameter server. It can reduce management overhead of the centralized structure and apply adaptive synchronization models for different parameter shards. Base on this design, we are able to propose overlap synchronization, which reduces up to 93.7% communication time costs in previous systems.
- We introduce lazy pull execution to reduce synchronizations overheads caused by frequent soft barriers and also guarantee robust convergence. When the server appends a pull request from the fast worker into a buffer, the buffer is set to be indexable via this worker's current progress. The server only executes the buffered pull and return its updated parameters after the slowest worker catches up.
- We improve convergence speed and test accuracy by running a probability-based synchronization model, namely Probabilistic Staleness Synchronous Parallel (PSSP), which is different from using deterministic synchronization under a bounded staleness/delay. Our algorithm can also determine the probability in both static and dynamic ways (e.g., sensitive to staleness and gradients). We provide theoretical analysis about the upper bounds of regret, which proves our model can suppress the accumulated errors caused by delayed gradients on robust convergence.

Table I: Synchronization model supports in different parameter server architectures.

	Synchronization Models
PS-Lite [4]	BSP, ASP, Bounded delay
Bösen [5]	BSP, SSP, ASP
STRADS [7]	Pipelining, Prioritization[1]
FlexPS [28]	BSP, SSP, ASP
Multiverso [29]	BSP, ASP
GeePS [30]	BSP, SSP, ASP
FluentPS	Flexible synchronization, e.g., BSP, SSP, ASP, DSPS, Drop stragglers, PSSP.

[1] STRADS's synchronization strategies work only for model parallelism.

II. Background and Motivations

A. Parameter Server Architecture

With the ever-increasing amount of labeled data, it usually takes days and weeks to train the deep neural networks (DNNs) for high accuracy [14, 30]. The approach that can accelerate the training process is to distribute the computational workloads to multiple nodes. Most distributed deep learning systems (e.g., TensorFlow [8], MXNet [4, 9]) are built on parameter server architecture, which provides push/pull operations for parameter synchronization between workers and servers. In a parameter server architecture, each worker iteratively calculates gradients and pushes them to servers to update the global model parameters; while a set of servers maintain these parameters and aggregate the gradients from all workers. With data parallelism, each worker trains a model copy on a subset of the training data; while using model parallelism, each worker trains a part of the model across the whole dataset. In the hybrid parallelism approach, the training data are partitioned and each part is assigned to a work group, which consists of several workers. Each work group trains the neural networks using model parallelism.

Table I shows comparisons of the key features of the state-of-the-art parameter server architectures. PS-Lite [4] applied a bounded delay model and its programming filters enabled users to design the synchronization models for different deep learning (DL) tasks. Bösen [5] supported SSP model (i.e., bounded staleness) when synchronizing the model parameters between faster workers and slow workers. It developed the SSP model via SSPtable, a table-based API, which recorded the staleness information and invalidated outdated parameters cached in workers. To provide flexible data parallelism control for machine learning (ML) tasks, FlexPS [28] introduced multi-stage abstraction. It divided one ML task into multiple stages, mapped each stage to an individual task, and dynamically adjusted parallelism degree according to the workloads of each stage. This optimization brings the efficient execution of an ML task with various workloads but fails to reduce synchronization overhead for stages with high data parallelism degree in most DL tasks. Moreover, these PS systems cannot change different synchronization models at runtime.

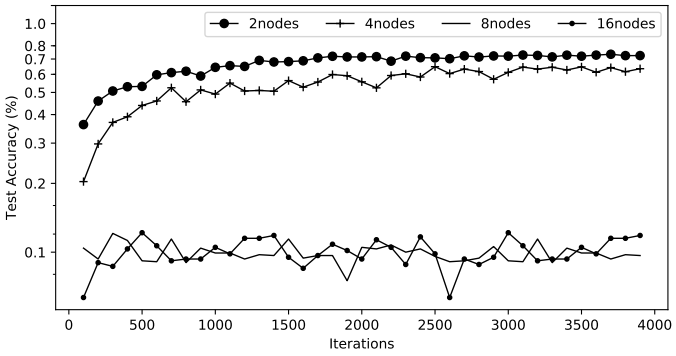


Figure 1: The test accuracy of AlexNet on CIFAR-10 with the same mini-batch size at different cluster scales. We run these experiments in PMLS-Caffe [31], an open-source Bösen implementation.

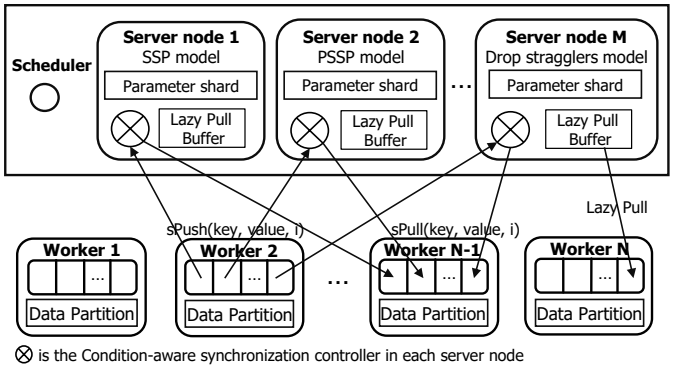


Figure 2: System architecture overview of FluentPS.

Table II: Explanations for the notations used in this paper.

N	The total number of workers is N.
M	The total number of parameter server is M.
n	The current worker is the n-th worker W_n.
m	The current server is the m-th server.
s	The predetermined staleness threshold.
w_i^n, $w_i^{n,m}$	$w_i^{n,m}$ are parameters pulled from the m-th server and used to calculate gradients in the i-th iteration on the n-th worker. We define $w_i^n = \cup_{m=1}^{M}(w_i^{n,m})$.
g_i^n, $g_i^{n,m}$	$g_i^{n,m}$ are gradients calculated in the i-th iteration on the n-th worker and pushed to the m-th server. We define $g_i^n = \cup_{m=1}^{M}(g_i^{n,m})$.

B. Motivations

There are several concerns to cause overheads in parameter synchronization and we list them below:

Limitations of current synchronization control: Previous parameter server frameworks exploit two ways to manage the parameter synchronization: (1) a centralized scheduler (e.g., PS-Lite [4]) records the progress of every worker and assigns one single synchronization model to each DL task, (2) each worker decides whether to block itself by comparing its progress with the version of parameters stored at server nodes (e.g., SSPtable [26], Angel [11]). Since the workloads of current DL tasks keep heavy, the data parallelism degree could be very large. The scheduler of PS-Lite [32], which uses one single model to control the synchronization of all parameters, can only achieve sub-optimization for reducing synchronization costs. On the other hand, without the scheduler, it is hard for a worker to detect the progress of other workers and many synchronization models (e.g, dropping stragglers) cannot be implemented, thus these systems lack flexibility.

Scalability vs. robust convergence: Some synchronization models (e.g., SSP [26]) need to track the progress of each worker about updating the global model parameter. With tracking more workers, the overhead to maintain a consistent parameter view in SSPtable becomes significant and causes poor scalability. For instance, we use Bösen to train AlexNet [33] on CIFAR-10 by using different worker sizes. When running the SSP model with the same staleness threshold, the 8-workers and 16-workers cases show less test accuracy (see Figure 1) than the 4-workers or 2-workers cases at the same iteration. We also observe similar or even worse convergence loss in other neural networks, which inhibits the benefits of large batch size. Current large-scale deep learning systems could use hundreds of GPUs (e.g., Facebook's Caffe2-based system [34] scaled to 256 NVIDIA GPUs) to accelerate the computation. There is a need to guarantee robust convergence when designing efficient synchronization methods.

Synchronization frequency: Existing synchronization models, e.g., SSP model, usually cause high-frequent synchronizations. Regardless of the significance of parameter updates, the soft barrier blocks the fastest worker once it exceeds the slowest worker s iterations, which creates many unnecessary waits. Furthermore, A trade-off in responding to each pull request is the time delay for waiting gradients updated by slow workers and the staleness of reading parameters. For instance, the soft barrier finishes once all workers are within a specified range s of the current iteration and it may return stale parameters to fast workers. Although the cost paid for each synchronization is light, the soft barrier has two shortages: (1) the fast worker cannot read the updated parameters, which hurts the convergence speed; (2) since the slowest worker is still following $s-1$ iterations behind the fast workers, it might repeatedly satisfy the SSP condition and the soft barrier will appear frequently.

III. Design

A. System Overview

Compared with previous parameter server systems, our design shows several advantages (see Figure 2): (1) condition-aware synchronization controller: each server can adjust the synchronization model by configuring the Push/Pull condition and our system overlaps the synchronization processes of different parameter shards to reduce communication time, (2) lazy pull buffer: we execute these delayed pull requests lazily to read the updated parameters and avoid frequent synchronization, (3) the optimized synchronization model, i.e., PSSP model used in the server node 2: it relaxes the pull condition of the SSP model to reduce the unnecessary waits of

the fast workers. Our system implementation is derived from PS-Lite [4], a open-source parameter server system [35] used by many famous deep learning systems, e.g., MXNet [9].

The scheduler only works for monitoring the liveness of servers and divides the whole key space [4] into several key ranges. Since each server is responsible for one key range, each push/pull request may access multiple key ranges to synchronize all parameters. In PS-Lite, the default slicing method [4] incurs load imbalances problem because it puts most parameters on one key range of a server. We design Elastic Parameter Slicing (EPS) to remap the original keys of the parameters to new keys, which divide the model parameters evenly on all key ranges. When the number of servers changes, EPS can also rebalance the workloads among the alive servers. Based on data parallelism, each worker computes the gradients on its data partition and synchronizes parameters with parameter servers via *sPush/sPull* operations. In the same iteration, each parameter server can choose the adaptive synchronization model to update its parameter shard. For instance, in Figure 2, server node 1 uses SSP model, server node 2 uses PSSP model, and server node M uses drop stragglers model. Some notations used in this paper are summarized in Table II.

Algorithm 1 Condition-aware methodology using lazy execution to control synchronization models

Max_Iter: total training iterations, $\mathbf{w_0}$: the initial parameters, **progress**: current progress reported by the worker via sPush/ sPull, $\mathbf{V_{train}}$: the overall training progress on this server's shard, e.g., the progress of slowest worker, **Count**[**i**]: the number of workers that finished pushing gradients in the i-th iteration.

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

1: Initialize kv = KV::Worker(), $w \leftarrow w_0$
2: **for** i from 0 to Max_Iter **do**
3: $g_i^n \leftarrow$step(w_i^n)
4: kv.sPush(key, g_i^n, i)
5: kv.wait(kv.sPull(key, &w_{i+1}^n, i))
6: **end for**

Parameter Server m = 1, 2, ..., M:

1: Initialize $w^m \leftarrow w_0^m$, $V_{train} \leftarrow 0$
2: **function** PULLHANDLER(key^m, &$w_{i+1}^{n,m}$, $progress$)
3: **if** ***PULL_con*** *is satisfied* **for** V_{train} **then**
4: $*w_{i+1}^{n,m} \leftarrow w^m$
5: server.response()
6: **else**
7: $callbacks[progress].push($
8: **function** LAZYPULL(key^m, &$w_{i+1}^{n,m}$)
9: $*w_{i+1}^{n,m} \leftarrow w^m$
10: server.response()
11: **end function**)
12: **end if**
13: **end function**
14: **function** PUSHHANDLER(key^m, $g_i^{n,m}$, $progress$)
15: $w^m \leftarrow w^m + g_i^{n,m}/N$
16: $Count[progress] \leftarrow Count[progress] + 1$
17: **if** ***PUSH_con*** *is satisfied* **for** V_{train} **then**
18: **for** Pull in $callbacks[V_{train}]$ **do**
19: Execute func(Pull)
20: **end for**
21: $callbacks[V_{train}].clear()$
22: $V_{train} \leftarrow V_{train} + 1$
23: **end if**
24: server.response()
25: **end function**

Table III: Equivalent implementation of flexible synchronization models by specifying push and pull conditions in Algorithm 1.

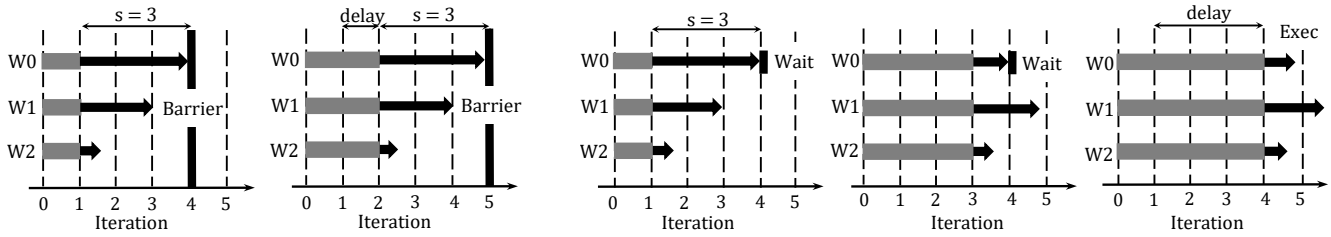
Model	Pull condition	Push condition
BSP	$progress < V_{train}$	$Count[V_{train}] == N$
ASP	$progress < V_{train} + \infty$	
SSP	$progress < V_{train} + s$	
DSPS		
Drop stragglers	$progress < V_{train}$	$Count[V_{train}] == N_t$
PSSP	$progress < V_{train} + s$ ***or*** $rand(0,1) > P$	$Count[V_{train}] == N$

B. Flexible Synchronization Model Control

In a large-scale distributed deep learning, N workers collaborate with M servers on the training process. We provide two extended operations, *sPush* and *sPull*, for the workers to synchronize different parameter shards with servers as well as report their progresses. Since each server stores only a part of the model parameters, the worker needs to communicate with multiple servers (i.e., W_n reads $w_i^{n,m}$ from the m-th server). There are several APIs to facilitate efficient parameter synchronization implementation. The SetcondPull function ($SetcondPull(keys, func = PULL\_con, argc, **argv)$) specifies the pull condition ($PULL\_con$). With proper pull condition, the faster workers could avoid reading stale parameters with some absent gradients. The SetcondPush function ($SetcondPush(keys, func = PUSH\_con, argc, **argv)$) specifies the push condition ($PUSH\_con$) that allows the server to execute delayed pull requests. These interfaces also expose details of the synchronization state, e.g., the progress of fastest/slowest worker, the number of workers that have pushed gradients in a specified iteration. The developers can create various synchronization models with respect to these states.

As Algorithm 1 makes clear, we present the detailed execution flow on the worker side and server side, respectively. In Table III, we give several examples of specifying the pull/push condition to easily implement various synchronization models, e.g., dropping slower workers [19] (all workers can enter the next iteration after the parameter servers receive gradients from any N_t workers) and adaptive staleness control [25].

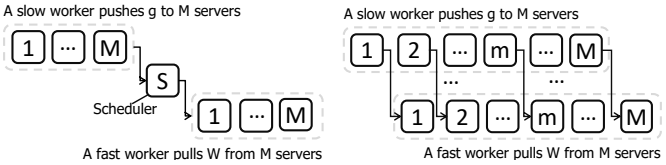
Worker side. All worker begin from the same initial model parameters (w_0). In the i-th iteration, worker$_n$ calculates the gradients (g_i^n, line 4) and sends g_i^n as well as its progress (i) to servers (line 5). It then waits for the accomplishment of



(a) Soft barrier: the parameters returned to W_0's pull are stale after W_2 completes the 2-th iteration (i.e., one iteration delayed). Frequent barriers (at the 4-th/5-th iterations) happen.

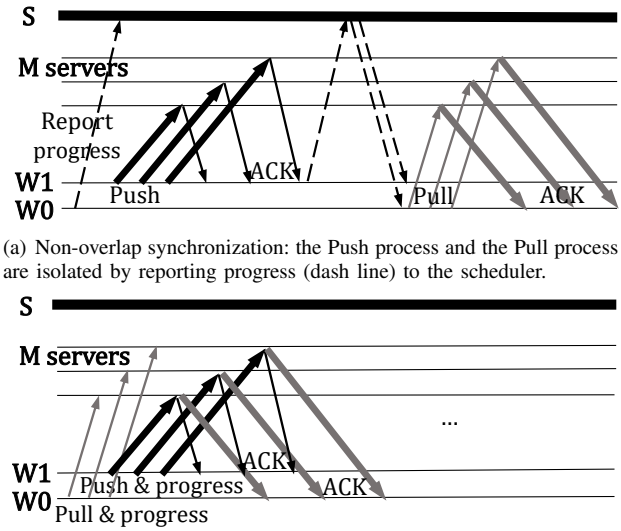
(b) Lazy execution: W_0 reads the updated parameters with only one pause (at the 4-th iteration). The server responds to W_0's pull after W_2 completes the 2-th, 3-th, and 4-th iterations (i.e., three iterations delayed).

Figure 3: The trade-off between the time delay of waiting for slow workers and the staleness of parameters that are returned to fast workers. From Iteration i to Iteration i+1, each worker W_n calculates the gradients g_i^n, pushes these gradients to servers and pulls the parameter w_{i+1}^n. The gray bar is the overall training progress (V_{train}), while the black arrow is the current progress of each worker.



(a) Non-overlap synchronization (b) Overlap synchronization

Figure 4: The comparison between the traditional designs and FluentPS about synchronization among different parameter shards.



(a) Non-overlap synchronization: the Push process and the Pull process are isolated by reporting progress (dash line) to the scheduler.

(b) Overlap synchronization: the Push process and the Pull process are overlapped. The progress of each worker is sent to servers.

Figure 5: The **time-line** diagram of two synchronizations: A pull (in gray) from a fast worker and A push (in black) from a slow worker. The thick line means it transfers the gradients/parameters; the fine line means it carries little data; the dotted line only appears in Non-overlap synchronization for reporting the progress to the scheduler.

reading the parameters (w_{i+1}^n) from servers (line 6) by telling what version of parameters that are needed for calculating gradients in the next iteration.

Server side. Based on the type of requests, our design classifies the synchronization conditions into pull condition and push condition. For each pull request to read a parameter shard, the server determines whether to return the current parameters or delay the response. If the pull condition (PULL_con) returns false, there are not enough workers updating their gradients to the server and the parameters are stale. The server appends this pull request into the lazy pull buffer (line 7-11), which is indexed by the progress of the worker that sends this pull. After applying each push request, the server counts the number of workers that finished pushing their gradients (line 16) and checks the push condition (PUSH_con). If it returns true, the server has aggregated enough gradients to increase V_{train} (the overall training progress on its parameter shard) and executes all delayed pull requests recorded in the lazy pull buffer (line 18-20).

C. Lazy Pull Execution

In Section III-B, the pull condition (server, line 3) requires the server to delay the response to this pull (we call it delayed pull request, DPR) from the fast worker and ensures the accuracy of convergence, while the push condition (server, line 17) determines how lazy to answer the delayed pull requests. In our algorithm, lazy execution uses *progress* (transferred via *sPull*) as the index (server, line 7) of the lazy pull buffer that records these DPRs, while the soft barrier uses V_{train}. In Figure 3, we illustrate the trade-off between the time delay of responding to each DPR and the staleness of parameters returned to this DPR when s equals 3. Before the slowest worker W_2 pushes g_1^2, the fast worker W_0 cannot pull w_4^0 because g_1^2, g_2^2, and g_3^2 are absent: (1) soft barrier (see Figure 3(a)): after one iteration delay and g_1^2 is updated to servers, the barrier finishes and W_0 reads w_4^0 without g_2^2 and g_3^2 (i.e., stale parameters), which could harm the robust convergence, (2) lazy execution (see Figure 3(b)): W_0 needs to wait until g_1^2, g_2^2, and g_3^2 are pushed to the servers (i.e., three iterations delay) and it can read the updated parameters.

D. Overlap Synchronization

Except for being more flexible, our system can reduce communication time via overlapping the push and pull processes. In non-overlap synchronization [4, 19, 32], the scheduler controls the parameter synchronization and it behaves like a

global barrier (the '**S**' in Figure 4(a)). It does not allow the fast workers [32] to send pull requests (see Figure 5(a)) until the slowest worker update all parameter shards. In FluentPS, the parameter synchronization models are independently controlled on different servers, instead of the single scheduler. As shown in Figure 4(b), once the slowest worker pushes its gradients to one server node, this server can immediately send its updated parameter shard to the fast workers, instead of waiting for the slowest worker to update other M-1 parameter shards. Compared with non-overlap synchronization in traditional designs (e.g., PS-Lite), our experiments show that overlap synchronization in FluentPS could cut down on the communication time by up to 93.7% in the case of using EPS to balance the workloads on the servers.

E. Probabilistic SSP Model

The motivation of the SSP model is to expect the randomly slow workers can catch up with the faster workers in the following few iterations, instead of synchronizing all workers in each iteration. There are two aspects to evaluate the efficiency of the SSP model: (1) synchronization frequency: how many DPRs are stored in the lazy pull buffer during training, (2) delayed gradients: how many gradients from the slow workers are missing for the servers to respond a pull. As a fundamental trade-off, the SSP model allows users to pre-determine a staleness threshold s that balances between the synchronization frequency and delayed gradients. A high staleness threshold can reduce the number of DPRs but some gradients may be seriously delayed; while a low threshold can guarantee timely parameter updates at the cost of extra synchronization overhead.

In this section, we propose Probabilistic Staleness Synchronous Parallel (PSSP) model, which relaxes the pull condition in SSP. Even the worker is s iterations faster than the slowest one, it will be blocked at a probability (P) before entering the next iteration. We further dynamically adjust this probability for different workers.

1) Constant PSSP model: The idea of constant PSSP is simple. We introduce a constant probability $P \in [0,1]$: the progress gap is less than s, P equals **0**; while the progress gap equals or is larger than s, P is c (c is a constant in this paper and $c \in [0,1]$) Especially, P equals 0, it becomes ASP model, while it reduces to SSP model if P is 1.

Theoretical Analysis. Similar to the proof of convergence bounds in SSP model, we focus on stochastic gradient descent (SGD) in this paper. Since different workers at different iterations have their own parameters (i.e., $T = Max\_Iter * N$), we collect them in a parameter sequence $W = \{w_1, w_2, ..., w_T\}$. As SGD searches w^* to minimize the cost function $f(w)$, $R[W]$ is the bound of the regret:

$$R[W] := f(w) - f(w^*), \text{ where } f(w) := \frac{1}{T}\sum_{t=1}^{T} f_t(w_t), w_t \in W$$

Proposition 1: **(SSPSGD [26]).** Under the following conditions: (1) $f_t(w_t)$ are L-Lipschitz functions, which are convex for any w and $||\nabla f_t(w_t)|| \leq L$ for some positive constant L; (2) For any w_1, $w_2 \in W$ and some positive constant F, $D(w_1||w_2)=\frac{1}{2}||w_1 - w_2||^2 \leq F^2$. Qirong Ho et al. [26] showed the bound of the regret, where N is the number of workers and s is the staleness threshold:

$$R[W](s,N) \leq 4FL\sqrt{\frac{2(s+1)N}{T}} \tag{1}$$

Theorem 1: **(Constant PSSP-SGD).** For each $w_i \in W$, we assume the progress gap between the worker pulls w_i and the slowest worker is s_i. We define $S = \{s_1, s_2, ..., s_T\}$ and $S' = \{s_i : s_i \geq s \;\&\&\; s_i \in S\}$. Based on description of constant PSSP, it can behave like the SSP model whose staleness threshold is s_i ($s_i \geq s$) at a probability of $c*(1-c)^{s_i - s}$. We can derive the bound of regret is $R[W] \leq \frac{1}{|S'|}\sum_{s_i \in S'} 4FL\sqrt{\frac{2(s_i+1)N}{T}}$, where the percentage of s_i appearing in S' is $c*(1-c)^{s_i-s}$:

$$\begin{aligned} R[W](s,N,c) &\leq \sum_{i=s}^{\infty} c*(1-c)^{i-s} * 4FL\sqrt{\frac{2(i+1)N}{T}} \\ &= \frac{4cFL}{(1-c)^s}\sqrt{\frac{2N}{T}}\sum_{i=s}^{\infty}(1-c)^i * \sqrt{i+1} \end{aligned} \tag{2}$$

To attain the upper bound of $A=\sum_{i=s}^{\infty}(1-c)^i * \sqrt{i+1}$, we have:

$$\begin{aligned} A^2 &= (\sum_{i=s}^{\infty} x^i * \sqrt{i+1})^2, \text{ where } x = 1-c \\ &\leq (\sum_{i=s}^{\infty} x^i) * (\sum_{i=s}^{\infty} x^i * (i+1)), \text{ from Cauchy inequality [36]} \\ &\leq (\sum_{i=s}^{\infty} x^i) * (\sum_{i=s}^{\infty} (x^{i+1})') = \lim_{n\to\infty}[\frac{x^s - x^{s+n}}{1-x} * (\frac{x^{s+1} - x^{s+n+1}}{1-x})'] \\ &\leq \frac{x^s}{1-x} * (\frac{x^{s+1}}{1-x})' = \frac{x^{2s}(s - s*x + 1)}{(1-x)^3} \end{aligned}$$

Under the same conditions of SSPSGD, the upper bound of Equation 2 can be expressed as:

$$\begin{aligned} R[W](s,N,c) &\leq \frac{4cFL}{(1-c)^s}\sqrt{\frac{2N}{T}}\sqrt{\frac{x^{2s}(s - s*x + 1)}{(1-x)^3}}, \text{ where } x = 1-c \\ &= 4FL\sqrt{\frac{2(c*s+1)N}{c*T}} = 4FL\sqrt{\frac{2(s+\frac{1}{c})N}{T}} \end{aligned} \tag{3}$$

Equation 1 and Equation 3 shows the regret's expectations of constant PSSP-SGD (s, c) and SSP-SGD (s'=s+$\frac{1}{c}$-1) could have the same upper bound: $4FL\sqrt{\frac{2(s+\frac{1}{c})N}{T}}$. As we will show in Section IV-B4, although the bounded regrets are equal, constant PSSP-SGD (with s, c) causes less frequency of synchronization: reduce up to 97.1% DPRs and 28.5% synchronization costs than SSP-SGD (with s'). Actually, the constant PSSP-SGD is more general than SSP-SGD and can provide fine-tuned staleness controls: s+$\frac{1}{c}$-1 can be any non-negative real number while s' is only a non-negative integer.

2) Dynamic PSSP model: Regardless of the progress, constant PSSP-SGD treats all fast workers equally by using one single constant probability P=c. For instance, the fastest worker exceeds the slowest worker s_1 iterations and another

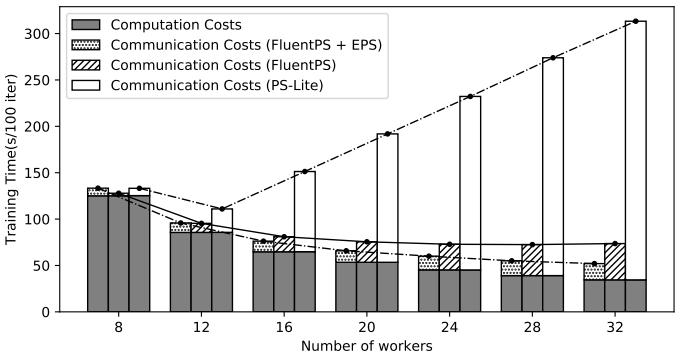


Figure 6: Computation/Communication time comparison of using FluentPS and PS-Lite to train ResNet-56 on CIFAR-10 (using BSP model) with different numbers of workers.

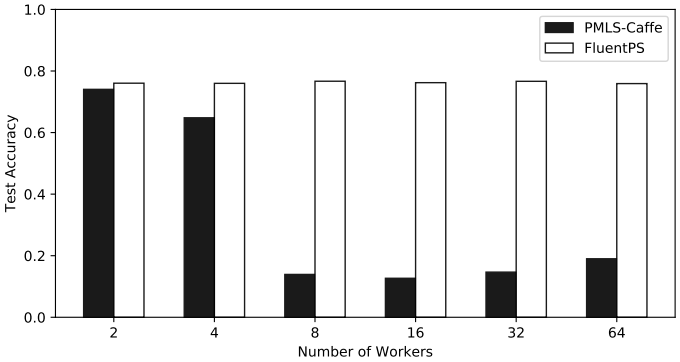


Figure 7: Test accuracy comparison of using PMLS-Caffe and FluentPS to train AlexNet on CIFAR-10 (using SSP model, s=3) with different numbers of workers at the 4000-th iteration.

fast worker exceeds the slowest worker s_2 iterations, assuming $s_1 \gg s_2 > s$. The probabilities of pausing these two faster workers are the same in constant PSSP, but the fastest worker should be blocked with a higher probability to inhibit accumulated errors on robust convergence because it reads very stale parameters. Therefore, it is sub-optimal to find a c in constant PSSP, which cannot be well adaptive to the pull requests from all fast workers. Dynamic PSSP calculates the probability for each worker based on its current progress, which both reduces synchronization overhead and ensures robust convergence.

Staleness consideration. Along with increasing the progress gap between the current worker and the slowest worker, it has a higher probability of pausing the current worker to wait for the gradients' updates from slow workers. The probability of buffering the pull request should be scaled up corresponding to the growth of the progress gap k and the staleness threshold s:

$$P(s,k) = \begin{cases} 0 & \text{If } k < s, \\ \frac{\alpha}{1+e^{s-k}} & \text{If } k \geq s. \end{cases}$$

α can be a initial threshold (i.e., a constant number) or a function (e.g., the significance function [37] $SF(g_i^n, w_i)$=$\frac{|g_i^n|}{|w_i|}$).

Theorem 2: **(Dynamic PSSP-SGD).** When α is a constant number, $P(s,k)$ is monotonically increasing function over the interval $[s,\infty)$. For each $s_i \in S'$, we calculate the corresponding probability $p_i = P(s,s_i) = \frac{\alpha}{1+e^{s-s_i}}$. We define $P = \{p_1, p_2, ..., p_{|S'|}\}$. Let $p_{min} = \min P = \frac{\alpha}{2}$ (i.e., s_i=s), and the regret of the dynamic PSSP-SGD is tighter than that of the constant PSSP-SGD when its constant probability equals $\frac{\alpha}{2}$. Based on Equation 3, we have $R[W] \leq 4FL\sqrt{\frac{2(s+\frac{2}{\alpha})N}{T}}$. When α is a function, the analysis should rely on the lower bound of this function. For instance, before the cost function reaches the local optimal solution, $|g_i^n|$ is larger than 0 and $\alpha = SF(g_i^n, w_i) > 0$.

IV. Experiments

A. Experimental Setup

Clusters setup. We evaluate the experiments on two clusters: (1) **Performance Test**: a GPU-cluster consists of 32 Amazon Elastic Compute Cloud (EC2) p2.xlarge instances and each of them is equipped with NVIDIA Tesla K80 GPU (12 GB GPU memory), Intel Xeon E5-2686 processors and 61 GB of RAM; they are connected via a 25 Gbps of aggregate network bandwidth. (2) **Scalability Test**: a CPU-cluster consists of 64 machines and each of them is equipped with two 4-cores Intel CPUs, 16GB DDR3 memory and 1Gbps Ethernet; they are connected via a 10 Gbps of aggregate network bandwidth. We further extend the number of workers to 128 by using Kubernetes in this cluster.

System setup. Each machine installs 64-bit Ubuntu 18.04 LTS and GPU instances use CUDA 10.0 and cuDNN toolkit 7.5. Without loss of generality [30, 37], we run one worker on each node of the CPU-cluster (a Caffe [10] process is one worker) and the GPU-cluster (a NVcaffe [38] process is one worker). In each cluster, we exploit different parameter server frameworks to update parameters with workers: FluentPS, PS-Lite [35] and PMLS-Caffe [31]. We use Layer-wise Adaptive Rate Scaling (LARS) [39] to support the large-batch training.

Models and datasets Our experiments use two datasets and two models. The first dataset for image classification, namely CIFAR-10 [40], consists of 60000 32x32 color images in 10 different classes, 50000 training images, and 10000 test images. The second dataset is CIFAR-100 [40] with "fine" labels, which has 100 classes and each class contains 500 training images and 100 testing images. To train these datasets, we respectively use AlexNet [33] and ResNet-56 [1] to achieve 76.5% accuracy and 93.2% accuracy on the CIFAR-10 test set, while 43.8% accuracy and 69.2% on the CIFAR-100 test set.

B. Performance Evaluation

1) Overlap synchronization:

In this section, we train ResNet-56 model for 64000 iterations on the GPU-cluster. The dataset is CIFAR-10 and the batch size is set to be 4096. To get rid of the effects caused by other factors, like the optimized synchronization models, we use the BSP model among workers in these experiments. With increasing the number of workers (N) from 8 to 32, the computation time decreases because each worker has fewer workloads of computation. We measure and

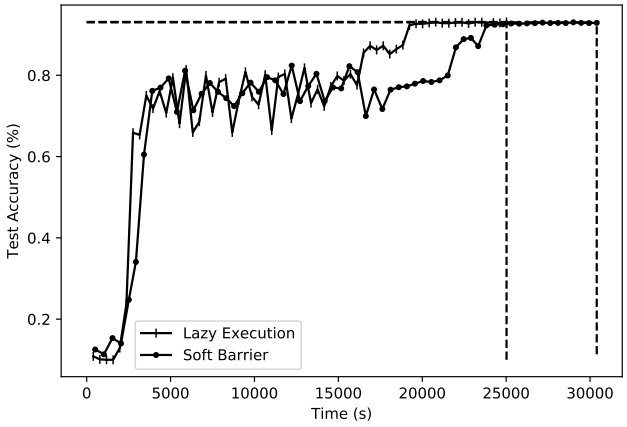


Figure 8: Test accuracy/Convergence speed comparison of using soft barrier and lazy execution to train ResNet-56 on CIFAR-10 with 32 workers and SSP model (s=2, totally 64000 iterations).

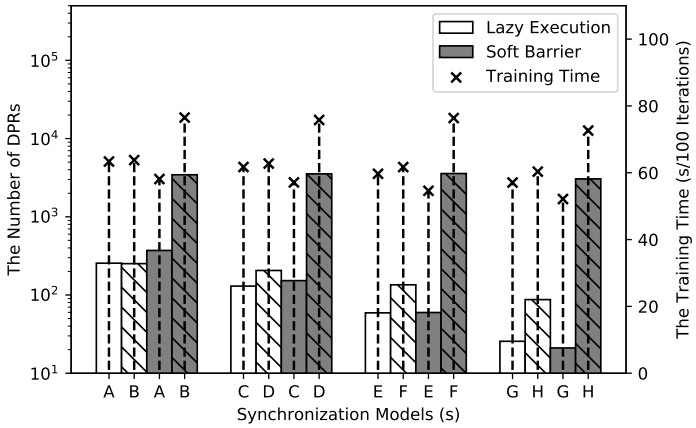


Figure 9: The number of DPRs per 100 iterations when training AlexNet on CIFAR-10: A, C, E, and G are PSSP models (s=3) with c respectively equals $\frac{1}{2}$, $\frac{1}{3}$, $\frac{1}{5}$, and $\frac{1}{10}$; B, D, F, H are SSP models with s respectively equals 4, 5, 7, and 12.

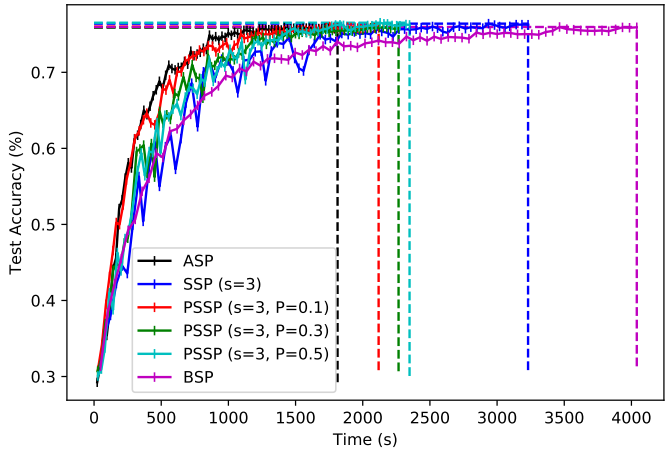


Figure 10: Accuracy vs. time for AlexNet on CIFAR-10 with 64 workers by using different synchronization models, totally 4000 iterations. PSSP (s=3, P=0.5) has the highest accuracy on testset of CIFAR-10.

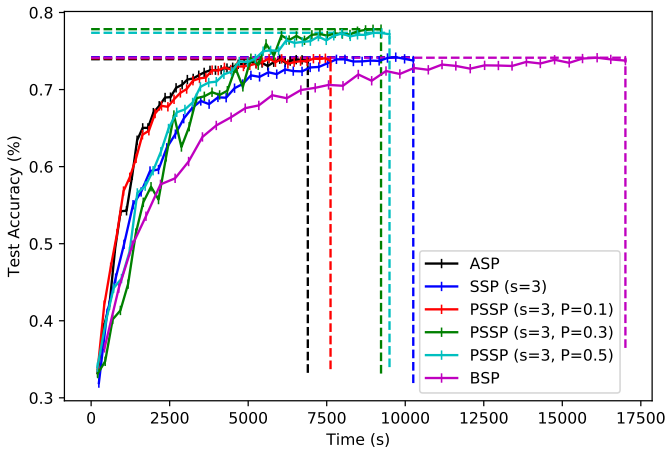


Figure 11: Accuracy vs. time for AlexNet on CIFAR-10 with 128 workers by using different synchronization models, totally 4000 iterations. PSSP (s=3, P=0.3) has the highest accuracy on testset of CIFAR-10. On top of our CPU cluster with Kubernetes v1.14.1, we launch a cluster of 128 instances and each runs one Caffe worker.

compare the results of three experiments in Figure 6: (1) PS-Lite (non-overlap synchronization): the communication time costs increased dynamically to dominate the total training time; (2) FluentPS (overlap synchronization): it can be up to 4.26× faster than PS-Lite and can reduce 86% communication time costs in non-overlap synchronization. (3) FluentPS with EPS, which balances the workloads of 8 servers: it can further achieve up to 1.42× speedup than FluentPS and reduce 55% communication time. In addition, overlap synchronization will not bring stale weights issue.

2) Scalability:

With 64-nodes CPU-cluster, We train AlexNet on CIFAR-10 dataset and the batch size is set to be 6400. Figure 7 shows the test accuracy of PMLS-Caffe and FluentPS after 4000 iterations. As we discussed in Section II-B, SSPtable in PMLS-Caffe could cause serious convergence loss when the number of workers becomes large, e.g., the test accuracy is less than 20% when N is larger than 8. The test accuracies of FluentPS are respectively 2.0% (on a 2-node cluster) and 11.2% (on a 4-node cluster) higher than those of PMLS-Caffe. When the cluster size increases to 64, FluentPS can still achieve 75.9%-76.7% accuracy on the CIFAR-10 test set, which is much higher than the accuracy (12.7%-19%) trained by PMLS-Caffe. This proves FluentPS is more scalable than SSPtable to support large deep learning systems with guaranteeing high accuracy.

3) Lazy execution vs. soft barrier:

In this section, we use different execution models to handle DPRs when training ResNet-56 on CIFAR-10. Figure 8 shows the test accuracy per 1000 iterations and the maximum number of iterations is set to be 64000. The synchronization model among 32 workers is SSP and s is set to be 2. Executing DPRs via lazy execution could be 1.21× faster than the soft barrier. Since the fast worker can read updated parameters

Table IV: Comparison among ASP (P=0), SSP (P=1), constant PSSP (P=0.1, 0.3, 0.5) and dynamic PSSP model. These models respectively use soft barrier and lazy execution and we show the average time and the number of DPRs per 100 iterations and the finial test accuracy.

			Soft barrier					Lazy Execution				
DNN	**P**	0	0.1	0.3	0.5	1	Dynamic	0.1	0.3	0.5	1	Dynamic
AlexNet	**Time**	45.85	52.95	56.67	58.71	80.77	72.55	58.85	60.96	63.89	64.62	62.44
on	**Acc**	0.759	0.760	0.759	0.765	0.764	0.762	0.756	0.756	0.762	0.758	0.765
CIFAR-10	**DPRs**	0	23.65	126.1	350.3	4002	2670	24.80	112.5	254.0	336.8	190.4
AlexNet	**Time**	48.84	51.00	54.38	57.59	78.68	71.81	56.83	59.43	61.97	62.48	65.73
on	**Acc**	0.429	0.429	0.426	0.430	0.432	0.438	0.430	0.431	0.434	0.434	0.437
CIFAR-100	**DPRs**	0	19.01	102.3	379.3	3834	2724	20.44	114.1	275.6	331.0	717.8
ResNet-56	**Time**	37.94	41.91	43.51	44.61	46.57	46.76	38.63	38.40	38.45	39.16	37.94
on	**Acc**	0.925	0.926	0.928	0.924	0.924	0.929	0.930	0.932	0.929	0.931	0.930
CIFAR-10	**DPRs**	0	906.4	3979	7462	15160	4539	40.28	32.46	56.77	115.1	49.62
ResNet-56	**Time**	37.07	42.02	46.99	47.27	48.79	45.83	37.64	37.88	37.07	37.31	37.57
on	**Acc**	0.685	0.687	0.689	0.688	0.686	0.688	0.688	0.690	0.692	0.689	0.689
CIFAR-100	**DPRs**	0	1085	5995	9993	21289	4092	32.91	82.31	83.56	77.50	26.48

* AlexNet is tested on our 64-node CPU Cluster (1 server and 64 workers, s=3); ResNet-56 is evaluated on the 32-nodes AWS GPU Cluster (8 servers and 32 workers, s=2).

in lazy execution, it shows more robust convergence. Lazy execution could convergence faster than soft barrier, e.g., synchronization with lazy execution has much higher accuracy from 16500s to 23500s in Figure 8.

As shown by the striped bars in Figure 9, we further evaluate the performance of training AlexNet with different staleness thresholds (4, 5, 7 and 12) in the SSP model. Compared with soft barrier, lazy execution can achieve up to 1.24× speedup by saving up to 97.1% DPRs during synchronization.

4) PSSP model:

In Section III-E1, we prove that the regret's expectations of PSSP-SGD (s, c) and SSP-SGD (s') have the same upper bound under some specified condition: s'=s+$\frac{1}{c}$-1. Therefore, we design four groups of experiments (i.e., A and B, C and D, E and F, G and H) via controlling s, c, and s'. The synchronization models in each group could share the same upper bound of regret. Figure 9 shows the PSSP model outperforms SSP by reducing up to 97.1% DPRs and 28.5% training time, i.e., model G vs. model H when using soft barrier. Under the premise of optimization brought by lazy execution, the PSSP can still save 70.7% DPRs in the SSP model. PSSP model with soft barrier sometimes has shorter execution time than lazy execution because PSSP can also decrease the number of DPRs and soft barrier needs fewer delayed iterations to execute each DPR.

In Figure 10, we compare BSP, SSP, ASP with PSSP (c equals 0.1, 0.3, 0.5 respectively) by measuring the parameters convergence progress of AlexNet on CIFAR-10. Although the ASP model is the fastest one to finish 4000 iterations, its test accuracy is the lowest among all models, which is around 1% lower than PSSP model (P=0.5). On the other side, the test accuracy of SSP model is close to PSSP model, while PSSP is 1.38× faster than the SSP model. We are further able to double the number of workers with 8 server nodes in Figure 11 by deploying Kubernetes to create a cluster of 128 container instances on our CPU clusters. In this case, PSSP model (P=0.3 or P=0.5) can achieve 3.9% higher test accuracy than the ASP model. Compared Figure 10 with Figure 11, we found the PSSP model shows its advantages of guaranteeing better test accuracy when increasing the number of workers.

Table IV presents extensive experimental results by evaluating different CNNs. Compared with shallow neural networks (e.g., AlexNet), lazy execution and PSSP model could cooperate better in training deeper neural networks like ResNet-56. The dynamic PSSP model relies on the configuration of α, which guarantees a high test accuracy or causes few DPRs among all models.

V. Related Work

A. Previous Parameter Servers

Most parameter servers [4, 11, 13] supported multiple synchronization models but they could only assign one model for executing one training task. FluentPS can adjust parameter synchronization model at runtime via controlling the push/pull conditions. To support synchronization models like SSP, Bösen used SSPtable [13], which is based on a convenient shared-memory model which invalid the outdated parameter entries cached at workers. Although the shared-memory model made it easy to program the distributed version of DL programs, it encountered serious scalability concern and might cause convergence loss. Our design adopts message passing and each worker reports its progress to servers and each server could synchronize its own parameter shard independently.

FlexRR [41] could control the workloads assigned to different workers based on their gradients calculation speed. For instance, the fast worker can help the slow worker to execute some parts of its work, instead of waiting for them to catch up. In their experiments, the authors admitted work reassignment could cause potential overhead when shifted workloads from stragglers. FlexPS [28] introduced a multi-stage method to split a whole task according to its dynamic workloads, such as SGD with growing batch size in ML applications. For each

stage of the task, it assigned the data parallelism degree to provide the trade-off between the computation time and the communication time. Different from common ML programs, our system targets large-scale DL systems, which have heavy workloads at every stage. Therefore, they still need very high data parallelism to minimize the per-iteration costs and the synchronization overhead among many workers still remains.

FluentPS also applies several approaches to improve the efficiency of PS, including overlap synchronization to reduce the communication time costs and lazy execution to do a trade-off between the short waiting time of replying to a DPR and robust convergence.

B. Synchronization Models in DL Research

BSP, ASP, and SSP. Bulk Synchronous Parallel (BSP) [16] blocked all workers to enter the next iteration until the slowest workers finished pushing its gradient to the parameter server. BSP was widely used in exiting PS systems, however, it suffered from the straggler problem which inhibits the training speed. Different from BSP, Asynchronous Parallel (ASP) [18] did not pause any faster workers to wait for the stragglers. Therefore, ASP had a higher training speed than BSP, but it could cause unexpected errors and the test accuracy might drop. Staleness Synchronous Parallel (SSP) [26] allowed delayed updates but the fastest worker could not exceed the slowest one more than a staleness threshold s. SSP provided a trade-off between BSP and ASP, which paused the faster workers if the progress gap among workers is no less than s. Especially, $s = 0$ means any worker need to wait for the others to finish the current iteration, SSP turns to be BSP; while $s = \infty$, SSP reduces to ASP.

There are many strategies to optimize SSP models with regard to learning rate schedules [21–23] and adaptive staleness [24, 25]. Dynamic Synchronous Parallel Strategy (DSPS) [25] monitored the performance of worker nodes to dynamically adjust the staleness threshold during the training process and therefore improved SSP. Probabilistic Synchronous Parallel (PSP) [42] was the first work to introduce probabilistic control in synchronization models, which effectively improved both training speed and scalability of systems. It created a boolean decision on whether or not to pass the synchronization barrier by sampling primitive, which derived a probability based on an estimation of all workers having pushed its gradients, e.g., a 10-node sample was taken from a cluster with 100s of nodes. However, the gradient significance from worker nodes are different, and we cannot treat them equally. The convergence loss will happen if the sampling workers have less gradient significance and their gradients are applied to the global parameters, while the delayed gradient updates in other workers are important but these workers are still regarded as finishing synchronization. Different from PSP, PSSP bases the probability calculation on the progress gap between workers and the gradients significance, which more adapt to synchronizing parameters in different training phases.

Others Synchronization Models. Some synchronization models focused on dropping the stragglers [19] and configuring the push/pull frequency of each worker [3]. To develop fast training and reduce frequency synchronization across datacenters, Gaia [37] introduced the significant filter to eliminate the network traffic of insignificant gradients when synchronizing parameter replicas across datacenters while still guaranteeing the correctness of convergence. It summarized that over 95% of updates produce insignificant gradient (e.g., less than 1% modification to the parameter value) and these gradients generated from several iterations can be aggregated and are not necessary to iteratively synchronized to the server of remote datacenters. SpecSync [43] was on top of ASP and SSP models and it allowed each worker to speculates about the recent updates from other workers. To synchronize some necessary updates, the worker needs to abort the calculation of current gradients and pull updated parameters from servers. Similarly, PSSP model can also determine the probability based on the quality of parameters but avoid the computation aborts in SpecSync model. Furthermore, the centralized scheduler was a bottleneck because it received the notifications from all workers after their push operations and implemented the logic of SpecSync on behalf of each worker. Our design offloads the synchronization model controls from the centralized scheduler on to server nodes, which can distribute the management overhead and overlap the gradients push processes and parameter push processes between workers and different server nodes.

VI. Conclusion

Along with increasing scales, there are many concerns in distributed deep learning: (1) previous parameter servers could cause significant communication overhead and convergence loss, (2) frequent synchronization happens when running existing synchronization models. Our paper proposes FluentPS, which introduces low-frequency but high efficient synchronization and provides several methods to optimize the communication costs. Some key findings are listed below:

- FluentPS uses overlap synchronization to reduce the waiting time in communication costs and EPS to balance the workload of servers. It achieves up to 6x speedup and reduces 93.7% communication costs than PS-Lite.
- Lazy pull execution returns the updated parameters to fast workers and guarantees robust convergence. It also achieves up to 1.24x speedup by bridging the progress gap among workers to reduce synchronization frequency.
- Under the SSP condition, FluentPS exploits PSSP model to pause the fast worker at a probability, which can be flexibly determined. It outperforms the SSP model by gaining 1.38x speedup and 3.9% higher test accuracy.
- FluentPS can well support large-scale distributed deep learning system because more workers will not cause convergence loss like PMLS-Caffe.

Acknowledgment

This research is supported by Hong Kong RGC grant 106160098 and AWS Cloud Credits for Research. We thank our shepherd and four anonymous reviewers for their suggestions, which help improve the quality of this paper.

REFERENCES

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016, pp. 770–778.

[2] G. Neuhold, T. Ollmann, S. R. Bulò, and P. Kontschieder, "The mapillary vistas dataset for semantic understanding of street scenes," in *2017 IEEE International Conference on Computer Vision (ICCV)*, Oct 2017, pp. 5000–5009.

[3] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large scale distributed deep networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12. USA: Curran Associates Inc., 2012, pp. 1223–1231.

[4] 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 *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, 2014, pp. 583–598.

[5] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Managed communication and consistency for fast data-parallel iterative analytics," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, ser. SoCC '15. New York, NY, USA: ACM, 2015, pp. 381–394.

[6] S. Lee, J. K. Kim, X. Zheng, Q. Ho, G. A. Gibson, and E. P. Xing, "On model parallelization and scheduling strategies for distributed machine learning," in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'14. Cambridge, MA, USA: MIT Press, 2014, pp. 2834–2842.

[7] J. K. Kim, Q. Ho, S. Lee, X. Zheng, W. Dai, G. A. Gibson, and E. P. Xing, "Strads: A distributed framework for scheduled model parallel machine learning," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16. New York, NY, USA: ACM, 2016, pp. 5:1–5:16.

[8] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283.

[9] 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," *CoRR*, vol. abs/1512.01274, 2015.

[10] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22Nd ACM International Conference on Multimedia*, ser. MM '14. New York, NY, USA: ACM, 2014, pp. 675–678.

[11] J. Jiang, L. Yu, J. Jiang, Y. Liu, and B. Cui, "Angel: a new large-scale machine learning system," *National Science Review*, vol. 5, no. 2, pp. 216–236, 2018.

[12] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu, "Petuum: A new platform for distributed machine learning on big data," *IEEE Transactions on Big Data*, vol. 1, no. 2, pp. 49–67, June 2015.

[13] 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 *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, 2017, pp. 181–193.

[14] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project adam: Building an efficient and scalable deep learning training system," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, 2014, pp. 571–582.

[15] S. Smith, P. jan Kindermans, C. Ying, and Q. V. Le, "Don't decay the learning rate, increase the batch size," 2018. [Online]. Available: https://openreview.net/pdf?id=B1Yy1BxCZ

[16] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.

[17] J. Albrecht, C. Tuttle, A. C. Snoeren, and A. Vahdat, "Loose synchronization for large-scale networked systems," in *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, ser. ATEC '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 28–28.

[18] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola, "Scalable inference in latent variable models," in *Proceedings of the Fifth ACM International Conference on Web Search and Data Mining*, ser. WSDM '12. New York, NY, USA: ACM, 2012, pp. 123–132.

[19] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous sgd," in *International Conference on Learning Representations Workshop Track*, 2016. [Online]. Available: https://arxiv.org/abs/1604.00981

[20] J. Cipar, Q. Ho, J. K. Kim, S. Lee, G. R. Ganger, G. Gibson, K. Keeton, and E. Xing, "Solving the straggler problem with bounded staleness," in *Presented as part of the 14th Workshop on Hot Topics in Operating Systems*. Santa Ana Pueblo, NM: USENIX, 2013. [Online]. Available: https://www.usenix.org/conference/hotos13/solving-straggler-problem-bounded-staleness

[21] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.

[22] J. Jiang, B. Cui, C. Zhang, and L. Yu, "Heterogeneity-aware distributed parameter servers," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17. New York, NY, USA: ACM, 2017, pp. 463–478.

[23] W. Zhang, S. Gupta, X. Lian, and J. Liu, "Staleness-aware async-sgd for distributed deep learning," in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, ser. IJCAI'16. AAAI Press, 2016, pp. 2350–2356.

[24] J. Hermans and G. Louppe, "Gradient energy matching for distributed asynchronous gradient descent," *CoRR*, vol. abs/1805.08469, 2018.

[25] J. Zhang, H. Tu, Y. Ren, J. Wan, L. Zhou, M. Li, J. Wang, L. Yu, C. Zhao, and L. Zhang, "A parameter communication optimization strategy for distributed machine learning in sensors," *Sensors*, vol. 17, no. 10, 2017.

[26] Q. Ho, J. Cipar, H. Cui, J. K. Kim, S. Lee, P. B. Gibbons, G. A. Gibson, G. R. Ganger, and E. P. Xing, "More effective distributed ml via a stale synchronous parallel parameter server," in *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'13. USA: Curran Associates Inc., 2013, pp. 1223–1231.

[27] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Exploiting bounded staleness to speed up big data analytics," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 37–48. [Online]. Available: http://dl.acm.org/citation.cfm?id=2643634.2643639

[28] Y. Huang, T. Jin, Y. Wu, Z. Cai, X. Yan, F. Yang, J. Li, Y. Guo, and J. Cheng, "Flexps: Flexible parallelism control in parameter server architecture," *Proc. VLDB Endow.*, vol. 11, no. 5, pp. 566–579, Jan. 2018.

[29] Microsoft, "Parameter server framework for distributed machine learning," https://github.com/microsoft/multiverso.
[30] 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 *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16. New York, NY, USA: ACM, 2016, pp. 4:1–4:16.
[31] sailing pmls, "Pmls-caffe: Distributed deep learning framework for parallel ml system," https://github.com/sailing-pmls/pmls-caffe.
[32] M. Li, "Synchronized sgd in ps-lite," https://ps-lite.readthedocs.io/en/latest/overview.html\#synchronized-sgd.
[33] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12. USA: Curran Associates Inc., 2012, pp. 1097–1105. [Online]. Available: http://dl.acm.org/citation.cfm?id=2999134.2999257
[34] P. Goyal, P. Dollá, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch sgd: Training imagenet in 1 hour," 2017. [Online]. Available: https://arxiv.org/abs/1706.02677
[35] M. Li, "A lightweight parameter server interface," https://github.com/dmlc/ps-lite.
[36] H.-H. Wu and S. Wu, "Various proofs of the cauchy-schwarz inequality," https://rgmia.org/papers/v12e/Cauchy-Schwarzinequality.pdf.
[37] 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 *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, 2017, pp. 629–647.
[38] borisgin, "Nvidia caffe (nvidia corporation ©2017) is an nvidia-maintained fork of bvlc caffe tuned for nvidia gpus," https://github.com/borisgin/nvcaffe.git.
[39] Y. You, I. Gitman, and B. Ginsburg, "Large batch training of convolutional networks," 2017. [Online]. Available: https://arxiv.org/abs/1708.03888
[40] A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009. [Online]. Available: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf
[41] 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 *Proceedings of the Seventh ACM Symposium on Cloud Computing*, ser. SoCC '16. New York, NY, USA: ACM, 2016, pp. 98–111. [Online]. Available: http://doi.acm.org.eproxy.lib.hku.hk/10.1145/2987550.2987554
[42] L. Wang, B. Catterall, and R. Mortier, "Probabilistic synchronous parallel," 2017. [Online]. Available: https://arxiv.org/abs/1709.07772
[43] C. Zhang, H. Tian, W. Wang, and F. Yan, "Stay fresh: Speculative synchronization for fast distributed machine learning," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, July 2018, pp. 99–109.