# Elastic Parameter Server Load Distribution in Deep Learning Clusters

Yangrui Chen
The University of Hong Kong
yrchen@cs.hku.hk

Yanghua Peng
The University of Hong Kong
yhpeng@cs.hku.hk

Yixin Bao
The University of Hong Kong
yxbao@cs.hku.hk

Chuan Wu
The University of Hong Kong
cwu@cs.hku.hk

Yibo Zhu
ByteDance Inc.
zhuyibo@bytedance.com

Chuanxiong Guo
ByteDance Inc.
guochuanxiong@bytedance.com

## ABSTRACT

In distributed DNN training, parameter servers (PS) can become performance bottlenecks due to PS stragglers, caused by imbalanced parameter distribution, bandwidth contention, or computation interference. Few existing studies have investigated efficient parameter (aka load) distribution among PSs. We observe significant training inefficiency with the current parameter assignment in representative machine learning frameworks (*e.g.*, MXNet, TensorFlow), and big potential for training acceleration with better PS load distribution. We design *PSLD*, a dynamic parameter server load distribution scheme, to mitigate PS straggler issues and accelerate distributed model training in the PS architecture. An exploitation-exploration method is carefully designed to scale in and out parameter servers and adjust parameter distribution among PSs on the go. We also design an elastic PS scaling module to carry out our scheme with little interruption to the training process. We implement our module on top of open-source PS architectures, including MXNet and BytePS. Testbed experiments show up to 2.86x speed-up in model training with *PSLD*, for different ML models under various straggler settings.

## CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**.


*SoCC '20, October 19–21, 2020, Virtual Event, USA*

ACM ISBN 978-1-4503-8137-6/20/10…$15.00
https://doi.org/10.1145/3419111.3421307

## 1 INTRODUCTION

Recent years have witnessed the rapid development of machine learning (ML) applied in various realms of business, science, and online services [11, 20, 27]. Due to increasing scales of data and training models, distributed ML frameworks, such as TensorFlow [8], MXNet [16] and PyTorch [32], have been widely applied to expedite model convergence with parallel training on multiple machines.

The parameter server (PS) architecture [28] is a popular paradigm for distributed DNN (deep neural network) training, widely supported in different ML frameworks. In the PS architecture, workers carry out model training and exchange their model updates through a set of parameter servers (PSs), which are responsible for maintaining different parameter chunks in the global DNN model. Compared to another synchronization paradigm, *i.e.*, AllReduce [41], the PS architecture adapts better in heterogeneous production data centers or public clouds, where the GPU clusters are usually connected with a large pool of CPUs and network bandwidth. Further, due to its flexible synchronization schemes and good support for both sparse training (*e.g.*, recommendation models) and dense training (*e.g.*, DNN models), many DNN model training workloads in production ML clusters adopt the PS architecture (*e.g.*, the production multi-tenant cluster used in our experiments).

The efficiency of workers and PSs affects training speed. Improving worker model training has been widely studied, in terms of mixed-precision training, graph compilation, and gradient coding [8, 17, 21, 35, 46]. On the other hand, the efficiency of PSs has been under-investigated. Straggler situations may well happen on the PS side as well, which significantly slows down the training progress (Sec. 2.3), especially for synchronous training jobs. For example, delayed parameter processing at the PSs prevents the workers from immediately executing the forward propagation of the next training iteration. The delay could be caused by many factors, *e.g.*, heterogeneous hardware, the contention of computation

and bandwidth on the PS side, and uneven parameter distribution among multiple PSs. In multi-tenant ML clusters where resource contention and performance interference are common [13], different PS processing speeds are especially the norm (Sec. 2.3).

In this work, we seek to accelerate distributed training by balancing PS load and mitigating PS stragglers. In current open-source ML frameworks (*e.g.*, MXNet and TensorFlow), parameter assignment among PSs is imbalanced and static, which does not change after training initialization (Sec. 2.2). While there have been research proposals [22][38] that study PS worker elasticity, they do not account for parameter load imbalance. Also, in the existing literature, very few studies have investigated PS load distribution. Optimus [35] partitions parameters larger than the average size and applies a descending best-fit algorithm for parameter assignment, whose performance is not compared against default ones in the ML frameworks. PS-Plus [3] considers the variance of PS performance and uses a simulated annealing algorithm to assign the parameters. These work do not consider parameter reassignment during training, while runtime straggler issue is often (Sec. 2.3).

Dynamic PS number adjustment during the training process is not supported in existing ML frameworks either. When some PSs become serious stragglers, we ought to remove them from working nodes; when the remaining available PSs are not sufficient for timely parameter update and exchange with workers, we shall increase PS nodes to distribute the model update/communication load. The common approach to changing the number of PSs in a training job is to do checkpoints of the training process [14, 35, 43], pause, and relaunch the job with new configurations. This introduces significant overhead into training. A more elastic, *hot* PS scaling approach is in need.

This paper proposes *PSLD*, a dynamic PS load distribution scheme to mitigate PS stragglers and identify a suitable number of PSs for the best training speed in distributed DNN training jobs. *PSLD* adopts an exploitation-exploration method to decide parameter assignment among PSs according to their performance, and dynamically adjusts the number of effective PSs on the go. Targeting elasticity and generality, we implement *PSLD* on the generic PS framework BytePS [5][26] (which supports popular distributed training frameworks including MXNet, TensorFlow, and PyTorch), as well as vanilla PS architectures such as MXNet PS.

We make the following contributions in developing *PSLD*:

▹ We observe significant training inefficiency with the current imbalanced and static parameter assignment in representative ML frameworks (MXNet and TensorFlow), and identify big potential for training acceleration with better PS load distribution in a multi-tenant cluster. We also measure the negative impact of PS stragglers and show that dynamic parameter reassignment can mitigate such issues and provide significant performance gain.

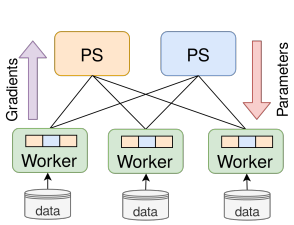



**Figure 1: Parameter server architecture.**

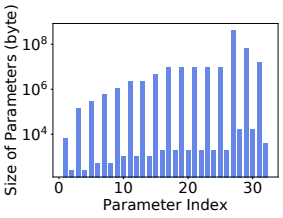



**Figure 2: Size (in log scale) of parameters in VGG16 model.**

▹ We propose a prediction-guided exploitation-exploration approach for dynamic PS load distribution. *PSLD* exploits historical profiling trace to assist in identifying a small set of good PSs. To alleviate inaccuracy in history-based prediction, it also randomly explores new PSs which can potentially have high performance, and maintains a good balance between exploitation and exploration. It assigns parameters to selected PSs with minimal costs, evaluated on normalized communication time and workload percentage, to effectively remove 'straggler' PSs from the working set. When the set of good PSs cannot provide good parameter update and communication performance, new PS nodes will be added.

▹ We further design an elastic PS scaling module and implement it on both vanilla MXNet PS and BytePS [5][26] architectures. It achieves dynamic parameter reassignment and PS scaling without checkpointing and restarting the training process, with negligible overhead.

▹ We evaluate our approach with testbed experiments training representative DNN models. The results demonstrate training speed-up up to 2.86x as compared to MXNet's default parameter distribution method, and 53% as compared to a strawman proportional parameter allocation approach under various PS straggler settings. *PSLD* also mitigates real-world straggler issues in the shared cluster and outperforms MXNet and TensorFlow frameworks by up to 49%. As compared to the checkpointing method, system overhead for PS elasticity scaling can be significantly reduced by over 90%, with *PSLD*.

## 2 BACKGROUND AND MOTIVATION

### 2.1 Inelastic Parameter Server

The PS architecture is widely adopted in today's distributed ML workloads, using an ML framework such as MXNet [16], TensorFlow [8] and Petuum [47]. In the PS architecture, parameters in the global ML model are partitioned among multiple PSs, and the training dataset is split among workers for data-parallel training (Fig. 1). In each training iteration,

workers compute parameter updates (*i.e.*, gradients) using its data partition, and push gradients to different PSs that maintain the respective model parameters. Each PS applies received gradients to its stored parameters with an optimization method, *e.g.*, Stochastic Gradient Descent (SGD) [39]. Then the workers pull updated parameters from the PSs for the next training iteration.

None of the existing (open-sourced) model training frameworks support elastic PS scaling and parameter reassignment during the training process. The common approach for the adjustment is to save the currently trained model definition and parameters as a checkpoint file and restart the training with a new PS configuration from the saved checkpoint [14, 35, 43]. However, using checkpointing to enable elasticity is not desirable, which involves library reload, model rebuilding and dataset preprocessing, with minutes of device idle time.

We would like to note that, in this work, we only consider stragglers that can affect PS. We discuss how stragglers can impact ML training workers in Section 8.

## 2.2 Imbalanced and Static Parameter Assignment

The parameter [1] assignment across PSs is often *imbalanced* and *static* in existing frameworks. TensorFlow [8] distributes parameters to multiple PSs in a round-robin way (in terms of parameter number); MXNet [16] splits parameters with large sizes (*e.g.*, larger than 4MB) and evenly distributes them to PSs, while randomly assigning smaller parameters among PSs. These default methods do not consider potentially vastly different sizes of parameters, or performance difference among PSs according to their available computation and communication capacities, in a shared cluster with resource contention. Static assignment strategies lead to vulnerability to runtime PS stragglers.

With the round-robin approach, sizes of parameters may well be imbalanced among PSs. Table 1 shows the parameter size on each PS when training the ResNet101 [23] model with different numbers of PSs. We observe highly imbalanced parameter sizes among PSs when the number of PSs is 3, 5, or 6. The situation is even worse when the sizes of parameters vary significantly, *e.g.*, in VGG16 [42] as shown in Fig. 2. The largest parameter tensor is over 400MB, while the total model size is 538MB.

Table 2 shows that the parameter size on each PS may also be imbalanced, with the parameter assignment approach in MXNet. It results from the random small parameter distribution in the scheme, which is in terms of parameter number but not size, and leads to varying overall parameter size on the PSs. To illustrate the impact of imbalanced parameter distribution, Fig. 3 shows the training speed at different numbers of PSs/workers, where we train each model with a worker-to-PS ratio of 1:1 (a common configuration among production DL workloads [35]) and homogeneous PS resource configuration on our testbed (Sec. 6). We observe obvious performance degradation when training ResNet101 [23] with 3 or 6 PSs and Inception_BN [24] with 4 or 6 PSs, due to imbalanced PS parameter distribution in these cases. Such imbalance significantly affects the linear scalability of training speed when we increase the number of PSs/workers in a training job. We observe similar issues when training other models, such as GoogLeNet [45] and VGG16 [42].

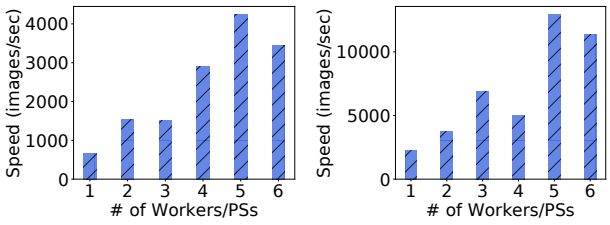
[1]The "parameter" refers to the learnable weights in each DNN layer.

**Table 1: Parameter allocation (number in millions) of ResNet101 on different numbers of PSs with TensorFlow**

| # of PSs | $PS_1$ | $PS_2$ | $PS_3$ | $PS_4$ | $PS_5$ | $PS_6$ |
|---|---|---|---|---|---|---|
| 1 | 44.5 | | | | | |
| 2 | 21.6 | 22.9 | | | | |
| 3 | 2.0 | **29.0** | 13.5 | | | |
| 4 | 9.9 | 10.3 | 11.7 | 12.7 | | |
| 5 | 8.5 | 7.1 | 9.9 | **12.2** | 6.8 | |
| 6 | 1.0 | **15.2** | 6.8 | 1.0 | 13.9 | 6.7 |

**Table 2: Parameter allocation (number in millions) of ResNet101 on different numbers of PSs with MXNet**

| # of PSs | $PS_1$ | $PS_2$ | $PS_3$ | $PS_4$ | $PS_5$ | $PS_6$ |
|---|---|---|---|---|---|---|
| 1 | 44.5 | | | | | |
| 2 | 22.3 | 22.3 | | | | |
| 3 | 7.5 | **31.1** | 5.9 | | | |
| 4 | 10.9 | 11.0 | 11.4 | 11.3 | | |
| 5 | 9.7 | 8.8 | 9.0 | 8.3 | 8.7 | |
| 6 | 3.7 | **15.6** | 3.0 | 3.8 | 15.6 | 2.9 |




**(a) ResNet101** **(b) Inception_BN**

**Figure 3: Training speed with different numbers of workers/PSs in MXNet.**

## 2.3 Runtime PS Stragglers

Even when parameter assignment is balanced, the performance of PSs varies and some PSs may become stragglers

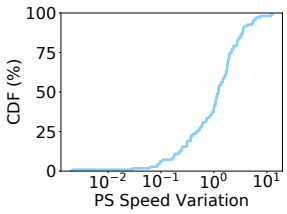



**Figure 4: CDF of PS speed variation (in log scale) of 100 DL jobs in a shared GPU cluster.**

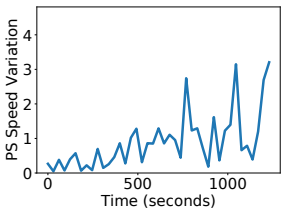
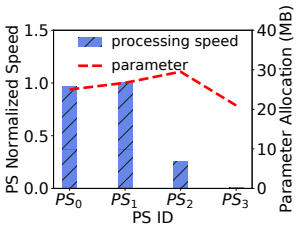
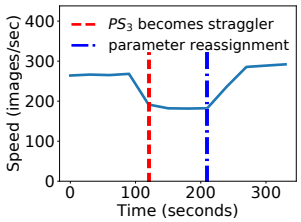



**Figure 5: Variation of PS speeds in a ResNet50 [23] training job over time.**




**(a) PS performance and parameter assignment (after $PS_3$ becomes a straggler).**




**(b) Training speed over time.**

**Figure 6: Training ResNet50 model with a PS straggler.**

during runtime because the current *static* parameter assignment cannot react to resource contention and interference in a shared cluster (*e.g.*, interference on computing resources and contention on shared NIC bandwidth). The performance degradation may be transient or persistent as jobs come and go.

We collect 100 DNN training jobs' traces from a production GPU cluster (see **Cluster-B** in Sec. 6), recording speeds of PSs, calculated by the total size of parameters a PS handles divided by the average gradient/parameter communication time between the PS and workers. The jobs run on BytePS with MXNet as the training framework. The PSs in a job run on the same hardware configurations (4 CPU-core container with 30GB memory). For each job, we compute the PS speed variation by $\frac{Speed_{best}-Speed_{worst}}{Speed_{worst}}$. Fig. 4 shows the CDF of PS speed variation of all jobs. We see that speeds of PSs in 60% of the jobs vary significantly (best PS is up to 10 times faster than worst PS).

In addition, the performance of individual PSs may vary significantly over time. Fig. 5 shows the change of speed variance among all PSs in a representative training job over time. The variance fluctuates significantly and frequently between 0.005 to 3.18.

Fig. 6 further illustrates the impact of varying PS performance when training the ResNet50 model using 4 PSs, where $PS_2$ has inferior performance than others starting from the beginning of training and $PS_3$ becomes a straggler during training due to bandwidth contention. The PS speed in Fig. 6(a) is normalized by dividing the maximal speed among all PSs. In Fig. 6(b), when $PS_3$ becomes a straggler, the overall training speed of the job drops significantly by up to 32%, as the response time for the parameter pull request from $PS_3$ becomes 33x more than the average response time from other PSs (as in Fig. 6(a)). A straightforward approach to mitigating the straggler issue is to reallocate parameters to PSs proportionally according to their performance. Fig. 6(b) shows that with this strawman parameter reassignment, the training speed can catch up again and is improved by up to 60% (as compared to without parameter reassignment).

Apart from training with BytePS, we made similar observations with jobs running directly on vanilla PS architectures (MXNet and TensorFlow). Our observations show that PS straggler issue is an inherent problem in existing ML systems, which is nonetheless barely explored. They further exhibit the significant potential of balanced, dynamic PS load distribution in accelerating distributed training, which has inspired our design of *PSLD*.

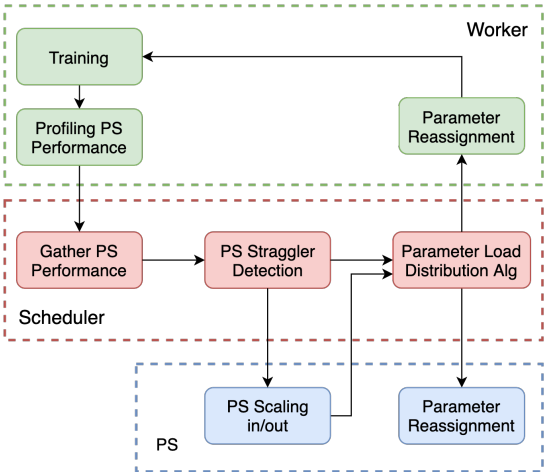



**Figure 7: *PSLD* workflow.**

## 3 SYSTEM OVERVIEW

Our goal in designing *PSLD* is to provide a resilient PS architecture and achieve expedited training, by dynamically adjusting parameter distribution among available PSs according to their runtime performance, and adding/removing PSs accordingly. The workflow of *PSLD* is shown in Fig. 7.

To achieve balanced parameter distribution in view of vastly different parameter sizes in a DNN (Sec. 2.2), we partition large parameters according to an empirical threshold

(*e.g.*, into 4MB blocks).[2] The partitioned parameter blocks are the units for parameter assignment in *PSLD*.

At the beginning of training, we assign partitioned parameter blocks in a round-robin manner to the PSs for a balanced load. During training, the workers monitor the performance of each PS by recording the communication time of each parameter (Sec. 4.1), and periodically report them to the *PSLD* scheduler (implemented as a coordinator of all workers and PSs). Based on the feedback from the workers, the scheduler checks if there is any straggler issue (Sec. 4) and then decides whether to reassign the parameters among the PSs. Once a PS straggler issue is detected, the scheduler applies the PS load distribution algorithm (*i.e.*, Alg. 3 in Sec. 4) to calculate the new parameter assignment among the PSs, and mitigate the influence of the straggler(s). The new parameter assignment will then be broadcast to all workers, and the workers will send gradients according to the new parameter assignment in the next training iteration.

The scheduler may decide not to deploy any parameters on particular PS(s); those PSs are essentially removed from the training job (scale-in of PSs). On the other hand, when the number of PSs with straggler issue is too large (*i.e.*, more than half of the initial number of PSs), the scheduler will add new PS(s) into the job (scale-out of PSs) if the addition of PS(s) leads to better training speed.

# 4 PS LOAD DISTRIBUTION

An efficient and dynamic PS load distribution method is in need, to mitigate the PS straggler issues and ensure high training performance at all times. A strawman approach is to reallocate parameters to PSs proportionally according to their profiled performance. However, such an approach highly relies on the profiling results and may easily abandon a transient PS straggler in the last scheduling round. To avoid falling into local optimum and achieve more effective load distribution, we design a prediction-guided exploitation-exploration approach.

Fig. 8 describes the procedure of our PS load distribution method. The scheduler periodically collects performance data of each PS from workers (stage 1) and adds them to history logs. With these data, the scheduler identifies if there exists a straggler issue: it checks the PS speed variance ($\frac{Speed_{best}-Speed_{worst}}{Speed_{worst}}$) among PSs; if the variance exceeds a threshold (*e.g.*, 1, as used in our experiments), the straggler issue exists. Then the scheduler forms a superior PS set by excluding slow PSs according to Alg. 1 (stage 2). In stage 3, the scheduler decides parameter assignment with an $\epsilon$-greedy policy [44]: exploitation of identified superior PS candidates

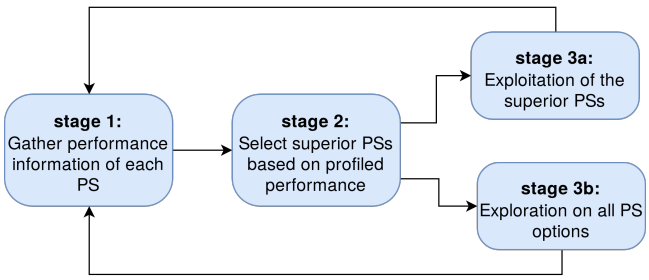



**Figure 8: Stages of *PSLD* dynamic PS load distribution algorithm.**

with probability $1-\epsilon$ (stage 3a) and exploration on all PS candidates with probability $\epsilon$ (stage 3b).

## 4.1 Performance Profiling

Each worker measures the performance of each PS $r$ in terms of its gradient/parameter communication time with $r$ and gradient/parameter size $s$: $Pred(r,s) = \alpha_r s + \beta_r$ [40], where $\alpha_r$ is the transmission time per byte of parameter, and $\beta_r$ is the latency component, including parameter update time and ACK time [3]. $\alpha_r$ and $\beta_r$ are decided by hardware configuration, bandwidth availability and interference at PS $r$, and are not related to parameter size $s$. The mean performance for a specific PS $r$ with parameter size $s$, $Pred_{mean}(r,s)$, and its standard error of mean (SEM), $Pred_{SEM}(r,s)$, are computed based on data collected from all workers.

## 4.2 Superior PS Set

Based on the performance measurements, we exclude a set of PSs with low performance and potentially being the stragglers, and obtain a superior PS set.

Considering the randomness in PS-worker gradient/parameter exchange time (due to variance in bandwidth availability and computation interference), we dynamically estimate the performance of each PS using a confidence bound instead of a fixed average value produced with collected data. Specifically, to identify the superior PS set, we compute lower and upper confidence bounds $Pred_{lower}(r,s)$ and $Pred_{upper}(r,s)$ of each PS's performance, where $Pred_{lower}(r,s) = Pred_{mean}(r,s) - 1.645 * Pred_{SEM}(r,s)$ and $Pred_{upper}(r,s) = Pred_{mean}(r,s) + 1.645 * Pred_{SEM}(r,s)$.[4] We find the superior PS set as the minimal set of PSs such that the lower 90% confidence bound of any PS not in the superior PS set is higher than the upper 90% confidence bound of any PS in the superior PS set. (Note that the lower the communication time with a PS is, the better the PS performs.) In other words, we are very sure that any PS that is not included in the superior PS set is worse

[2] This is different from MXNet's partitioning strategy, which divides the parameters larger than a threshold evenly among PSs, such that the parameter size at each PS can be much smaller than the threshold.

[3] We adopt a linear regression method based on the collected statistics.

[4] Our observed profiling results follow a normal distribution, and 1.645 reflects the 90% confidence level [18].

**Algorithm 1** Produce superior PS set

1: **function** GetSuperiorPS(Pred, $ps\_set$)
2: $SuperiorPS = \emptyset$, $RemainedPS = ps\_set$, $threshold = \infty$, $s = \frac{model\ size}{\#\ of\ PS}$
3: **while** *true* **do**
4: $r = \arg\min_{r \in RemainedPS} Pred_{upper}(r, s)$
5: **if** $Pred_{lower}(r, s) \geq threshold$ **then**
6: **break**
7: **else**
8: $threshold = Pred_{upper}(r, s)$
9: $SuperiorPS = SuperiorPS \cup r$
10: $RemainedPS = RemainedPS \setminus r$
11: **if** $|SuperiorPS| < ScaleThreshold$ **then**
12: Start scaling-out process
13: **return**
14: **else**
15: **return** $SuperiorPS$

**Algorithm 2** Exploit on the superior PS set

1: **function** Exploit($param$, $SuperiorPS$, $Assign$, Pred)
2: $cost_{min} = \infty$, $r_{best} = null$
3: $S_{assign} = \sum_{p \in Assign} p.size()$
4: $\omega = \frac{1}{|SuperiorPS|} \sum_{r \in SuperiorPS} Pred_{mean}(r, s)$ where $s$ is the total size of parameters on PS $r$
5: **for** $r \in SuperiorPS$ **do**
6: $C_r = \{c | Assign(c) = r\} \cup \{param\}$
7: $S_r = \sum_{c \in C_r} c.size()$
8: $cost = \frac{\theta_0}{\omega} Pred_{mean}(r, S_r) + \theta_1 \frac{S_r}{S_{assign}}$
9: **if** $cost \leq cost_{min}$ **then**
10: $r_{best} = r$
11: $cost_{min} = cost$
12: **return** $r_{best}$

than any PS that is included. For instance, the probability for the PS achieving the minimal communication time to be included in the superior PS set is more than 90%.

The procedure is given in Alg. 1 (lines 3-10), where we add the PS into superior PS set one by one according to its confidence intervals (line 5), and update a *threshold* according to the maximum upper confidence bound of the PS in the latest superior PS set (line 8). The gradually increasing *threshold* is used to cluster the superior PSs within 90% confidence level.

## 4.3 Exploitation and Exploration

We assign (partitioned) parameter blocks one by one to selected PSs. For each unassigned parameter block, we select one PS from the superior PS set according to Alg. 2 with probability $1 - \epsilon$; and select one PS to try in the entire set of

**Algorithm 3** $\epsilon$-greedy load distribution

**Require:** Parameters *params* to be assigned, available PS set $ps\_set$
**Ensure:** Assignment decisions $Assign$ of all parameters *params*
1: $Pred$ = ProfilePS($ps\_set$)
2: $Assign = \emptyset$
3: $SuperiorPS$ = GetSuperiorPS($Pred, ps\_set$)
4: **for** $param \in params$ **do**
5: **if** $Rand() \leq (1 - \epsilon)$ **then**
6: $Assign[param]$ = Exploit($param, SuperiorPS, Assign, Pred$), and put $param$ on the corresponding PS
7: **else**
8: $Assign[param]$ = Random($ps\_set$), and put $param$ on the corresponding PS

PSs with probability $\epsilon$, to avoid getting stuck into a local optimum. $\epsilon$ is a constant (set to 0.1 as default in our experiments) and remains unchanged during training. The complete PS load distribution algorithm is summarized in Alg. 3.

Our exploitation in Alg. 2 selects the best PS in the superior PS set with the minimal cost. The cost includes (i) the predicted mean communication time between workers and the PS, and (ii) the workload on the PS. The communication time is predicted according to the profiled performance model, as $Pred_{mean}(r, S_{size})$, where $S_{size}$ is the total size of parameters on PS $r$ based on the current parameter assignment (including the parameter to be assigned). The workload on a PS is modeled as $\frac{S_{size}}{S_{assign}}$, the percentage of parameters assigned to PS $r$ where $S_{assign}$ is the total size of parameters that have been assigned to all PSs. The cost (line 8 in Alg. 2) combines both factors with weights $\theta_0$ and $\theta_1$, while normalizing the communication time term with $\omega$: $\omega$ is the average of mean communication time among the superior PS set, such that the value is between 0 and 1; $\theta_0$ is the weight on the predicted communication latency for the assigned parameters and $\theta_1$ penalizes excessive assignment on this PS (we will evaluate different ratios of $\theta_0$ and $\theta_1$ in Sec. 6).

## 4.4 PS Scaling

When the PS straggler issue happens in large scale, although our algorithm in Alg. 3 strives to find the best assignment policy over all available PSs, it may still not be sufficient to mitigate the straggler impact due to the lack of available PSs. In this case, the scheduler adds some new PSs to share the workload of the existing PSs. Specifically, we enable scale-out of PSs when the number of PSs in the superior set is less than $ScaleThreshold$ (line 11 in Alg. 1, set to $\frac{1}{2}$ by default in our

experiments), by launching one more PS (line 12 in Alg. 1), *i.e.*, *ScaleThreshold* is the minimum size of the superior set.

# 5 SYSTEM DESIGN AND IMPLEMENTATION

We design *PSLD* to be generic over multiple ML frameworks and PS libraries. *PSLD* is implemented using C++11 on BytePS, exploiting its compatibility with different ML frameworks (*e.g.*, TensorFlow, PyTorch, MXNet), as well as directly on a traditional ML framework, *i.e.*, MXNet with the PS architecture (referred to as vanilla MXNet PS).

BytePS is an open-sourced generic communication library, which adopts Horovod's interface to interact with multiple ML frameworks and replaces the NCCL communication library [7] with PS library for gradient aggregation. Different from traditional PS architecture, BytePS does not maintain global model parameters in the PSs; the PSs are used purely for gradient aggregation in each training iteration and send aggregated gradients to workers, and workers use them to update their model parameters. In this way, it saves the cost for various optimizers' initialization and parameter updates for PS nodes in different ML frameworks. To embed our dynamic parameter reassignment in BytePS, we do not need to migrate parameters when parameter assignment changes, but synchronously redefine the mapping from each gradient to the responsible PS; while in the implementation on MXNet PS, we move parameters across PSs in case of parameter reassignment.

The details of our PS scaling and parameter assignment are described below.

## 5.1 Dynamic PS Scaling

For a running job, no existing distributed ML framework supports dynamic scaling, *i.e.*, adding or removing PSs during training. The key challenge is how to efficiently re-distribute gradients to PSs, as well as adjust the inter-connection among the nodes. We carefully design and implement a dynamic scaling approach to adjust PS deployment.

We demonstrate our PS scaling procedure using the example of adding a new PS into an existing job, which can be divided into 3 steps, as shown in Fig. 9.

**1) Initialization.** When a new PS node is launched, it registers itself with the scheduler by sending an "ADD_SERVER" request message. After the scheduler accepts this request, it assigns an ID to the new node and responds to the request together with all workers' connection metadata (*e.g.*, IPs and ports). After that, the PS starts functioning, awaiting workers' gradients and further instructions from the scheduler (*e.g.*, gradient reassignment).

**2) PS load distribution.** When the scheduler receives a registration request, except for sending back the response to the new PS, it will also predict the network performance of the newly added PS using the average performance of the existing PSs, and run the load distribution method (Alg. 3 in Sec. 4) to reschedule the gradient assignment based on the current status. The new gradient assignment balances the load among the PSs according to each PS's performance periodically reported by workers, so as to optimize the job performance. The scheduler then sends the scaling command as well as the new PS's connection metadata (*e.g.*, IP and port) to all the workers.

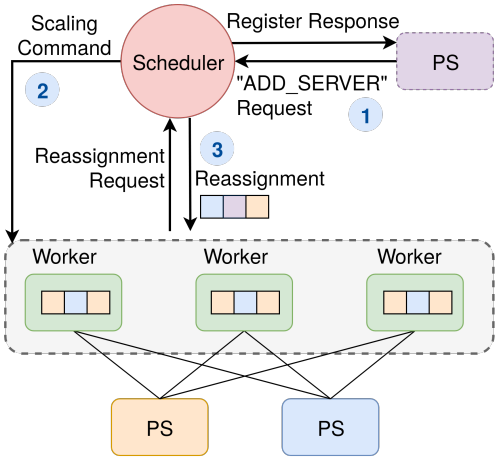



**Figure 9: Procedure of adding one PS into a distributed training job. It includes 3 steps in total. The arrows are the message flows between the scheduler, PSs and workers.**

**3) Gradient reassignment.** At each worker, upon receiving the scaling command from the scheduler, the worker blocks its training threads at the beginning of the new iteration, updates the new PS's information, and sends a gradient reassignment request message to the scheduler. After receiving the request messages from all workers, the scheduler starts parameter movement process among PSs (needed only for *PSLD*'s implementation on vanilla ML framework but not on BytePS) and broadcasts a notification to all workers for training resumption afterwards. The workers then adjust their gradient dispatch among PSs using received new assignment from the scheduler and unblock the training threads. The details of the gradient reassignment process will be introduced in Section 5.2.

In case of removing a PS, the PS to be removed sends a "DEL_SERVER" request to the scheduler. Similar steps as 2) 3) above are then carried out, and the parameters/gradients that the removed PS was responsible for are reassigned to other PSs, based on the load distribution algorithm (Alg. 3) by the scheduler.

Our scaling design can handle cases that server shutdown is notified beforehand. For unexpected faults, *PSLD* still rely on periodic parameter checkpointing. PS replication can mitigate the straggler problem to some extent, but incur significant extra resource consumption; besides, the replicas themselves may become stragglers.

## 5.2 Dynamic Reassignment

In existing distributed ML frameworks, parameter distribution among all PSs is determined and remains unchanged after initialization. To support parameter reassignment for a running job without training termination, we add extra interactions in different PS architectures.

**Vanilla ML framework.** Upon receiving the new reassignment from the scheduler, PSs need to migrate parameters. To retain a consistent copy of global model parameters during the migration process, we maintain a version counter for parameters. To decide when PSs should migrate parameters, the scheduler also broadcasts a reassignment step number to all PSs, which is calculated based on the current version counter and round trip time between the scheduler and PSs/workers. At each PS, when the version counter of parameters reaches the reassignment step, the PS moves its parameters to the destination PSs according to the parameter assignment decisions received. Once parameter migration among all PSs is completed, the scheduler notifies all workers to resume training.

**BytePS.** On BytePS architecture, we add interactions with BytePS core and low-level communication library (*i.e.*, PS-Lite), as shown in Fig. 10. As briefly mentioned in 5.1, the scheduler broadcasts a reassignment response to all workers, and the workers notify the upper layer, BytePS core, to block the training thread at the beginning of the next iteration. Then the BytePS layer informs PS-Lite [2], a light and efficient implementation of the PS framework, to be ready to request new gradient distribution among PSs from the scheduler. The scheduler responds with gradient reassignments to workers after receiving the requests from all workers, which maintains training synchronization and gradient distribution consistency for all workers. The workers update the gradient-PS mapping using received reassignment strategies and unblock the BytePS core to continue training.

The overhead of parameter migration across PSs, as incurred in the dynamic reassignment stage, is negligible. For BytePS implementation, PSs are used for gradient aggregation only and do not store parameters. There is no gradient or parameter migration across PSs, and hence the parameter reassignment overhead is zero. For vanilla ML framework, the overhead is negligible as parameter migration overlaps with parameter pull response.

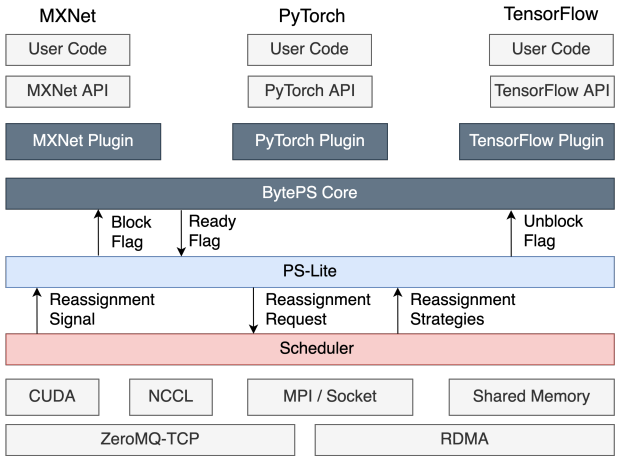



**Figure 10: Steps for gradient reassignment in an existing job. The dark grey parts are BytePS components. The arrows denote the interactions among components.**

# 6 EVALUATION

## 6.1 Methodology

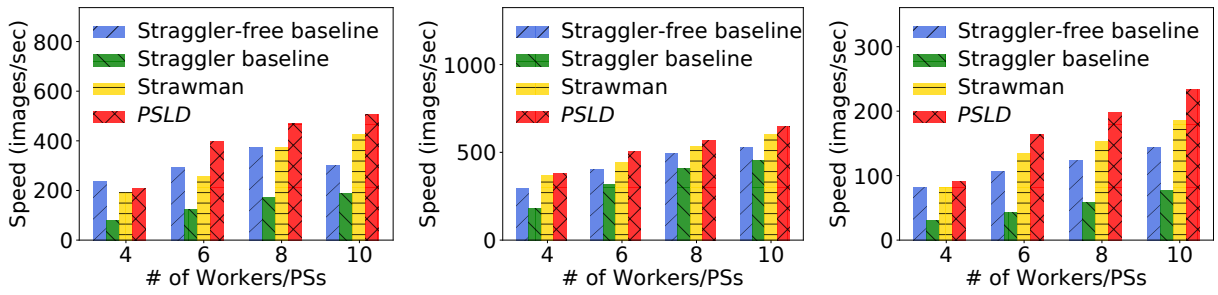
**Testbed.** We employ two clusters for our evaluation experiments:

• **Cluster-A** includes 8 GPU servers connected by a Dell Networking Z9100-ON switch with 25Gbps peak bandwidth between any two servers. Each server has one 8-core Intel E5-1660 CPU, two GTX 1080Ti GPUs, 48GB RAM, one MCX413A-GCAT NIC, one 480GB SSD, and one 4TB HDD, and installs NVIDIA GTX driver 384.90, CUDA 9.0, CuDNN 7.0 and NCCL 2.4.7 together on Ubuntu 14.04 LTS.

• **Cluster-B** is a multi-tenant production cluster with hundreds of servers, each equipped with 64 CPU cores, 320GB memory, 8 Tesla V100 GPUs *without* NVLinks, and 100Gbps bandwidth between any two servers using Mellanox CX-5 single-port NICs. There are typically hundreds of ML training jobs running concurrently in this cluster.

**Workload.** We evaluate *PSLD* by training 3 representative models, *i.e.*, AlexNet [27], ResNet50 [23] and VGG16 [42], with their default training datasets [1] in the MXNet framework. Each worker occupies 1 GPU, and the batch sizes per GPU for the 3 models are 32, 32, 64 samples, respectively. Each PS runs on a 4 CPU-core container with 30GB memory. Upon the start of a training job, we use a worker and PS ratio of 1:1, and run in synchronous mode on our testbed.

**Thresholds.** By default, the parameter partition unit is 4MB; we set $\epsilon = 0.1$ (see Alg. 3) and $\frac{\theta_1}{\theta_0} = 1$ (see Alg. 2). We will conduct sensitivity experiments on these threshold/variable values.




(a) AlexNet (b) ResNet50 (c) VGG16

**Figure 11: Training speed with different numbers of workers per job with MXNet.**

**Comparison.** We compare our dynamic PS load distribution approach with the static load distribution method of vanilla MXNet and a strawman proportional reassignment method:

- *Baseline* (which is the default parameter assignment method in MXNet): It splits the parameters with large sizes (>4MB) evenly among all PSs and randomly distributes smaller parameters among PSs.
- *Strawman*: It reallocates parameters to PSs proportionally according to their profiled performance.

To evaluate PS scaling and parameter reassignment overhead, we use the checkpointing method as a baseline, which is supported in most existing distributed ML frameworks.

**Metrics.** We use the training speed (images/sec) as the main performance metric. PS scaling and parameter reassignment overhead is evaluated using the suspension time that the procedure brings.

**Straggler patterns.** We produce straggler patterns as follows, together with extreme patterns providing stress tests:

- *Slow PS pattern*: We limit the maximum bandwidth of certain PSs using Linux tc tool [6]. With such rate control, communication with slow PSs results in longer transmission time.
- *Disrupted machine pattern*: We emulate computing resource contention by running a disruptive process occupying CPU cores on some PS machines, executing an intensive computation loop. These PSs will have less average CPU cores due to the disruptive process, resulting in delays for aggregating gradients from workers.
- *Real-world pattern*: This is the varying PS performance in the multi-tenant production **Cluster-B** with resource contention and interference as we showed in Fig. 4 and Fig. 5.

We use *straggler intensity%* to denote the percentage by which a PS's bandwidth or computation capacity is lowered.

By default, evaluation results are from the BytePS-based implementation (using MXNet as the training framework), and results from *PSLD*'s implementation on vanilla MXNet PS are specified.

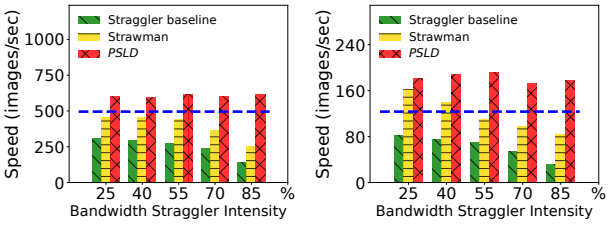
## 6.2 PS Straggler Mitigation

We first train each DNN model using a 10Gbps network setting (available bandwidth is limited by Linux tc tool) to emulate the typical bandwidth settings in AWS cluster (*e.g.*, g3.4xlarge GPU instance) [4], on our own testbed **Cluster-A**.

**Scalability vs. speed.** We limit the network bandwidth of one PS to 1Gbps to produce a PS straggler and train each DNN model using different numbers of workers and PSs. Fig. 11 compares the training speed of baseline (default method in MXNet) without any PS straggler, baseline with 1 bandwidth-limited slow PS, strawman proportional reassignment and *PSLD* (both with the PS straggler) with different starting numbers of workers/PSs. We observe that our dynamic load distribution method can outperform MXNet's default parameter distribution (when PS straggler issue occurs) by 39%-286%, the straggler-free baseline by up to 68% and the strawman reassignment by up to 53% across the 3 benchmark models.

Fig. 11 further shows that the PS straggler can cause significant performance downgrade, especially in communication-intensive models (AlexNet and VGG16). Although there only exists 1 slow PS, the training speed in AlexNet, ResNet50 and VGG16 can decrease by up to 67%, 38%, and 62%, respectively, with static parameter allocation. By dynamically balancing the load among all PSs, our approach can greatly mitigate the problem and achieve improvement of 224%, 109%, and 286% for these 3 models, respectively.

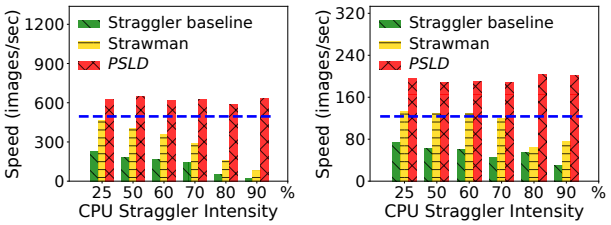
The improvement of *PSLD* over straggler-free baseline is mainly due to the default unbalanced parameter assignment of the baseline method (Sec. 2.2), which leads to skewness in parameter distribution among PSs and hence nonlinear scalability of training speed. Besides, the commonly used 1:1 configuration of worker and PS numbers is often non-optimal, especially when there are more than 8 workers [35]. By selecting the superior PS set, our approach reassigns most




(a) ResNet50 (b) VGG16

vspace-2mm

**Figure 12: Training speed with different limited bandwidth of PS stragglers (2 stragglers): 8 workers per job. The dotted blue line denotes the results of straggler-free baseline.**



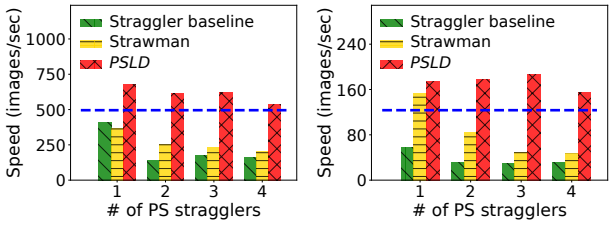

(a) ResNet50 (b) VGG16

**Figure 13: Training speed with different average CPU resources of PS stragglers (2 stragglers): 8 workers per job. The stragglers are set according to the disrupted machine pattern.**

of the parameters to PSs with the best performance, *i.e.*, the number of effective PSs in use can be less than the number of workers. In this way, *PSLD* further outperforms straggler-free baseline by up to 68%, 30% and 62% for the 3 models, respectively.

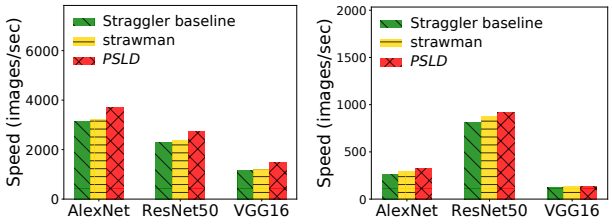
*PSLD* can also perform better than the strawman proportional assignment, which reassigns parameters to PSs proportionally to their performance. The performance of strawman does not differ much with *PSLD* when the number of workers/PSs is small, but the difference increases when the number grows, especially with the VGG16 model. Up to 53%, 13%, and 29% performance gain can be achieved with our approach over the strawman reassignment for the 3 models. The main reason lies in that we mostly exploit a set of superior PSs, while the strawman approach always uses the same number of PSs as workers. Besides, due to its proportional assignment, the strawman approach is very sensitive to the accuracy of the profiled performance of PSs, while our approach is more robust.

**Straggler variants vs. speed.** Fig. 12, Fig. 13 and Fig. 14 show the training speed of straggler baseline, strawman method and *PSLD* under different straggler intensities.




(a) ResNet50 (b) VGG16

**Figure 14: Training speed with different numbers of PS stragglers (1Gbps limited bandwidth for PS stragglers): 8 workers per job.**




(a) BytePS (b) MXNet PS

**Figure 15: Training speed with real-world PS stragglers on BytePS and MXNet PS: 8 workers per job.**

We first fix the numbers of slow PSs to 2, and vary the limited bandwidth of straggling PSs and the average CPU resources in Fig. 12 and Fig. 13, respectively. We see that the larger the straggler intensity is, the smaller the training speed of the baseline method achieves, while *PSLD* performs steadily and achieves up to 4.72x and 5.96x speed-up in the worst cases in these two settings, respectively. In contrast with *PSLD*, the strawman method is sensitive to the straggler intensity (bandwidth and CPU) and has a poor performance when the PS straggler issue is severe (up to 86% reduction compared to *PSLD*).

We also evaluate *PSLD* with different numbers of PS stragglers in Fig. 14. The baseline and strawman methods perform significantly worse as the number of stragglers increases, while the performance of *PSLD* is quite stable. With 4 slow PSs, *PSLD* can mitigate the PS straggler issue and achieve 3.4x and 5.5x speed-up for ResNet50 and VGG16, respectively, as compared to the baseline and 1.7x and 2.8x, as compared to the strawman approach.

**Real-world straggler pattern.** We next perform experiments on the multi-tenant **Cluster-B**, to evaluate *PSLD* with real-world PS straggler effects, which were observed consistently during our experiments (see Fig. 5). We do not inject any synthetic PS straggler effects in these experiments, and the average size of the superior PS set in the experiment is 7.

a) *PS architectures vs. speed.* Fig. 15 shows that *PSLD* can expedite training on both BytePS and vanilla MXNet PS architectures. *PSLD* on BytePS produces obvious speed-up (Fig. 15(a)) in the real-world environment, 18.4%, 19.5% and 26.1% respectively for AlexNet, ResNet50 and VGG16, as compared to the baseline. The improvements are produced despite rapidly-varying performance of the PSs (due to interference by other concurrent jobs), resulting from exploiting superior PSs to reduce stragglers' influence on training and the exploration step for avoiding falling into local optimum.

We also observe good speed-up when *PSLD* is implemented on MXNet PS architecture (Fig. 15(b)): 25.3% for AlexNet, 12.7% for ResNet50 and 10.9% for VGG16 (as compared to the baseline). On the other hand, the strawman approach cannot well handle such complicated straggler patterns, and no obvious speed-up is observed as compared to the baseline. *PSLD* outperforms the strawman by up to 21.6% on BytePS and 9.7% on MXNet PS.

In general, *PSLD* can obtain more gains with BytePS than with vanilla MXNet PS: PS straggler mitigation by *PSLD* can better exploit acceleration techniques used in BytePS, including hierarchical gradient synchronization (NCCL for local communication among GPUs in the same machine and push/pull for remote communication) and credit-based preemptive scheduling [5][26][37]. Nonetheless, *PSLD* can benefit vanilla PS architectures as well.

b) *Comparison with TensorFlow parameter assignment.* We further compare *PSLD*'s performance with TensorFlow (using its default round-robin parameter assignment) in **Cluster-B**. As shown in Fig. 16, *PSLD* outperforms TensorFlow by up to 49% across the three models.

**Dynamic PS scaling.** We evaluate the performance of our dynamic PS scaling approach when new PSs are added to existing jobs after detecting more than half of the PSs are slow (Sec. 4.4). In Fig. 17, 6 out of the 8 PSs become stragglers (bandwidth limited to 1Gbps) at 120 seconds, which leads to a sharp decrease of training speed; after detecting the straggler case, the scheduler launches the PS scaling process to launch new PS(s), and the training speed can be boosted again.

We further compare *PSLD* with the baseline, the strawman approach, and *PSLD* without the PS exploration module (only retaining exploitation) in the case of transient PS stragglers. As shown in Fig. 18, two PS stragglers appear at 90 seconds (bandwidth limited to 1Gbps) and recover at 360 seconds (bandwidth restored to 10Gbps). After parameter reassignment, *PSLD* achieves the best training speed recovery/improvement, which is 2.31x of the baseline. After the PS stragglers recover and another round of parameter reassignment, *PSLD* further improves the training speed and outperforms the strawman approach by 34%. This is because

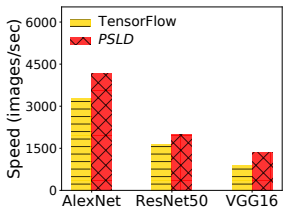



**Figure 16: Training speed comparison with TensorFlow parameter assignment: 8 workers per job.**

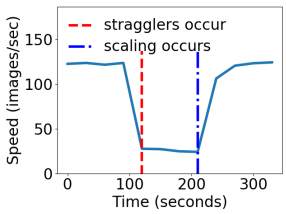



**Figure 17: Training speed (VGG16) with dynamic PS scaling: 6 slow PSs out of 8 PSs; 8 workers.**

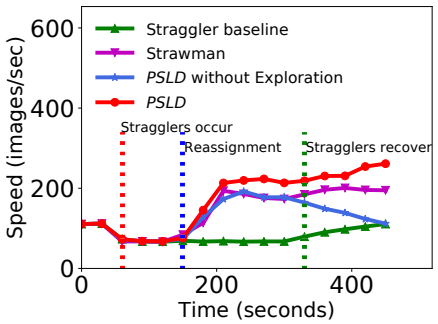
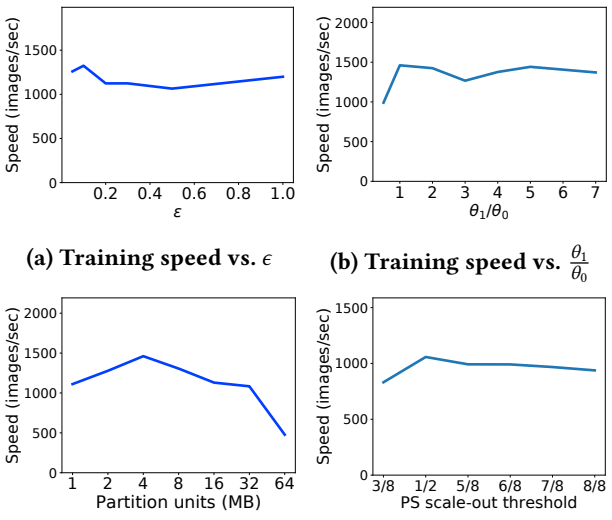



**Figure 18: Training speed with transient PS stragglers: 4 workers per job; training ResNet50.**

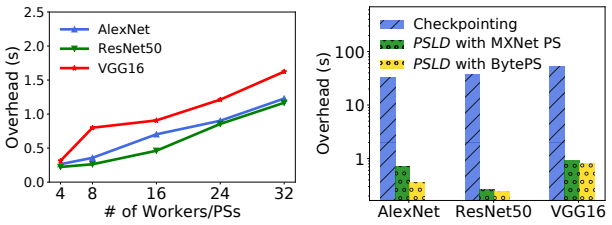
the strawman approach does not handle transient straggler situations well: it may assign only a few parameters to straggler PSs, so that the measured communication time with those PSs is largely affected by noise and hence not accurate, which prevents the strawman approach from reallocating more parameters to them when they recover. We compare the profiling performance of transient PS stragglers between the strawman approach and *PSLD*, and find the former more unsteady due to less profiling samples (assigned parameters). Without the exploration step, *PSLD* fails to identify the recovered PSs as well, which results in dropped performance over time. The dropped performance of the exploration-free *PSLD* is because it cannot sense straggler recovery, resulting from no parameter assigned in the last reassignment, and the size of the superior PS set tends to decrease with more reassignment procedures, leading to a heavy burden on individual PSs. It is noteworthy that the speed after the straggler recovery is larger than that before stragglers appear, which is due to the improvement of parameter distribution (originally skewed) and better configuration of worker vs. PS numbers (originally 1:1) after *PSLD*'s reassignment process.




(a) Training speed vs. $\epsilon$ (b) Training speed vs. $\frac{\theta_1}{\theta_0}$

(c) Training speed vs. parameter partition size (d) Training speed vs. *ScaleThreshold*

**Figure 19: Sensitivity experiments for variables in *PSLD*: training VGG16 in Cluster-B, 8 workers/PSs initially.**




(a) Overhead vs. job scale (b) Comparison with checkpointing (in log scale): 8 workers per job

**Figure 20: Overhead of the PS scaling and parameter reassignment process.**

## 6.3 Sensitivity and overhead Analysis

**Sensitivity.** Sensitivity experiments for variables in *PSLD*'s algorithms are conducted: a) $\epsilon$, which determines the probability for exploration among all PSs or exploitation (1-$\epsilon$) in the superior PS set (Alg. 3); b) $\frac{\theta_1}{\theta_0}$, which represents the trade-off between predicted communication latency and parameter assignment size in cost calculation of superior PSs (Alg. 2); c) parameter partition unit size; d) PS scale-out threshold (*ScaleThreshold* in Alg. 1).

As shown in Fig. 19(a), the training speed improves when $\epsilon$ is smaller than 0.1, and decreases by up to 24.3% as $\epsilon$ exceeds 0.1, due to exploring potentially inferior PSs with a larger probability. Fig. 19(b) suggests similar results with $\frac{\theta_1}{\theta_0}$, where the training speed peaks at $\frac{\theta_1}{\theta_0} = 1$, and fluctuates when its value increases.

Fig. 19(c) indicates that the best partition unit is 4MB in our experiments. Smaller partition unit size incurs more overhead in communication (not fully utilizing the bandwidth), while a larger block size reduces the scheduling space of our *PSLD* algorithm, leading to large speed decline. As shown in Fig. 19(d), the training speed first rises as the PS scale-out threshold increases and reaches the peak at 1/2; further increasing the threshold does not improve the performance, due mainly to additional communication overhead with more PSs.

**Overhead analysis.** The overhead of dynamic PS scaling and parameter reassignment process in *PSLD* is shown in Fig. 20. The overhead is evaluated by the suspended training time during the process.

Our dynamic PS scaling suspends training in the third step (*i.e.*, gradient reassignment in Sec. 5) and hence its overhead is equal to the time for applying parameter reassignment in a running job. The majority of parameter reassignment time comes from the execution of our load distribution algorithm and the synchronization of workers. Fig. 20(a) shows that the extra time cost of *PSLD* varies with different numbers of workers/PSs and models, but all smaller than 2 seconds. Larger models or more PSs lead to larger interaction time among the scheduler, workers and PSs in the process of dynamic reassignment.

In Fig. 20(b), we compare the overhead of *PSLD* with the checkpointing approach for PS scaling and parameter reassignment. *PSLD* incurs more than 90% less delay for both MXNet PS based and BytePS based implementations as compared to the checkpointing method due to its elastic scaling process, which saves the cost for restarting (including PS nodes initialization, graph building, etc.).

# 7 RELATED WORK

## 7.1 Elasticity in Distributed Learning

In recent years, a few elastic PS architectures have been proposed [22][38][36]. Proteus [22] adopts an elastic PS framework to exploit transient revocable resources and to reduce training cost in a public cloud. It observes the cloud's state and defines three transition stages with certain thresholds to elastically control the number of PSs and workers with minimal use of more costly non-transient resources. Litz [38] also targets at adapting to the changing resource availability, where machines can be added or removed during the execution of an ML application. The scheduler in Litz leverages elasticity for faster job completion and more efficient resource allocation. Proteus and Litz focus on exploiting dynamic resources in clusters but tend to replace slow workers with new ones when encountering straggler issues, while

our system explores the stragglers in a finer granularity and can handle transient stragglers without simply shutting them down. Further, Proteus and Litz do not identify the PS straggler issue, nor consider PS parameter adjustment.

Resource elasticity has also been explored for AllReduce architecture. Or et al. [31] present the autoscaling engine for distributed learning on TensorFlow with Horovod AllReduce communication library [41]. It detects straggler issues by comparing each worker's throughput with a predefined threshold, and uses its scaling mechanism to replace the detected straggler. However, this work does not investigate the multi-tenant use case in depth, and focuses more on elasticity than performance degradation. Besides, its simple straggler mitigation method may not be able to handle complex straggler patterns, such as transient stragglers.

### 7.2 Stragglers in Distributed Model Training

Many studies address worker stragglers in distributed ML. Some advocate less strict parameter synchronization to mitigate the synchronization cost, such as Stale Synchronous Parallel (SSP) [28], Dynamic Synchronous Parallel [48] and Round-Robin Synchronous Parallel [15]. AD-PSGD [29] and Hop [30] are variations of asynchronous and stale synchronous training, which target communication efficiency in heterogeneous environments. Some other studies focus on heterogeneity-aware distributed SGD algorithms, employing a constant learning rate for delayed gradient push in the SSP protocol, to reduce disturbance and unstable convergence caused by stragglers [25]. Such approaches may not provide guarantees of training convergence, and can affect model quality (accuracy).

Some adopt work stealing and work shedding methods to move workload from slow workers to fast workers [9, 21]. SmartPS [19] leverages the central control on the PS and prioritizes parameter transmission and update selectively and proactively on the side of PS, to narrow down the gap between straggler workers and fast workers. Blacklisting [10] is a performance variation elimination method, mitigating straggler issues by ceasing to assign work to slow workers. Instead of workers, we focus on the stateful PSs and address PS straggler issues with the proposed load distribution method.

AllReduce methods [12, 33, 34] also lack the flexibility to tackle straggler issues, which is more challenging than PS architecture due to the more restrictive communication pattern between workers. There are some works focusing on straggler issues in AllReduce [29, 30], but their methods may lead to deadlocks and may not be able to deal with complex straggler patterns, such as transient stragglers.

## 8 CONCLUDING DISCUSSIONS

We present *PSLD*, an elastic PS load distribution scheme to mitigate PS straggler issues and accelerate distributed model training. *PSLD* dynamically reassigns parameters across PSs according to an exploitation-exploration method, and adjusts the number of PSs accordingly. We implement *PSLD* on BytePS and vanilla MXNet PS architectures to support elastic PS scaling and parameter reassignment on the go. Our testbed experiments show that *PSLD* can improve training speed by up to 2.86x as compared to MXNet's default parameter distribution method, and 53% as compared to a strawman proportional assignment method under various controlled PS straggler settings. *PSLD* also mitigates real-world straggler issues in the multi-tenant production cluster and outperforms default parameter assignment approaches in MXNet and TensorFlow by up to 49%. It enables effective dynamic PS scaling in a running job, introducing less than 10% overhead as compared to the classical checkpointing method.

*PSLD* focuses on synchronous training so far, where all workers train a new mini-batch simultaneously and update the model based on the gradients in the same iteration. Synchronous training is the dominant deep learning mode in production machine learning clusters, given that asynchronous training may not ensure model convergence or accuracy. Our approach and implementation can be extended to, as well as benefit asynchronous training jobs as well. The major challenge is how to ensure correct gradient aggregation in cases of parameter reassignment. A possible and practical way is to maintain iteration steps at each worker and decide reassignment steps for different workers using each worker's own iteration steps and estimated training speeds during the scheduler's parameter reassignment process.

*PSLD* targets at mitigating PS-side straggler issues, and hence assumes that no slowdown on the worker side. Slow workers will affect the profiling accuracy of PSs and incur larger overhead in the process of reassignment due to larger synchronization time. *PSLD* can adopt existing worker straggler mitigation methods [21, 25, 28] to alleviate the influence of slow workers, which is orthogonal to our design on PS straggler mitigation.

Additionally, based on *PSLD*, there are many other research directions to explore in the future, *e.g.*, extending *PSLD* with fault-tolerant training capability, considering machine monetary cost or resource quota when scaling in/out PSs in multi-tenant clusters, and diagnosing and interpreting the root cause of stragglers.

## ACKNOWLEDGEMENTS

This work was supported in part by grants from Hong Kong RGC under the contracts HKU 17204619 and 17208920.

# REFERENCES


[1] 2012. ImageNet Dataset. http://www.image-net.org/.

[2] 2014. ps-lite: A Lightweight Parameter Server Interface. https://github.com/dmlc/ps-lite.

[3] 2019. Alibaba PS-Plus. https://github.com/alibaba/x-deeplearning/tree/master/xdl/ps-plus.

[4] 2019. AWS EC2 Instance. https://aws.amazon.com/ec2/instance-types/.

[5] 2019. BytePS: A High Performance and General Framework for Distributed Training. https://github.com/bytedance/byteps/.

[6] 2019. Linux tc. https://linux.die.net/man/8/tc.

[7] 2019. NCCL. https://developer.nvidia.com/nccl.

[8] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A System for Large-Scale Machine Learning. In *Proc. of USENIX OSDI*.

[9] Umut A Acar, Arthur Charguéraud, and Mike Rainey. 2013. Scheduling Parallel Programs by Work Stealing with Private Deques. In *Proc. of ACM SIGPLAN Notices*.

[10] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. 2013. Effective Straggler Mitigation: Attack of the Clones. In *Proc. of USENIX NSDI*.

[11] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. In *Proc. of ICLR*.

[12] Yixin Bao, Yanghua Peng, Yangrui Chen, and Chuan Wu. 2020. Preemptive All-reduce Scheduling for Expediting Distributed DNN Training. In *Proc. of IEEE INFOCOM*.

[13] Yixin Bao, Yanghua Peng, and Chuan Wu. 2019. Deep Learning-based Job Placement in Distributed Machine Learning Clusters. In *Proc. of IEEE INFOCOM*.

[14] Yixin Bao, Yanghua Peng, Chuan Wu, and Zongpeng Li. 2018. Online Job Scheduling in Distributed Machine Learning Clusters. In *Proc. of IEEE INFOCOM*.

[15] Chen Chen, Wei Wang, and Bo Li. 2019. Round-Robin Synchronization: Mitigating Communication Bottlenecks in Parameter Servers. In *Proc. of IEEE INFOCOM*.

[16] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2016. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. In *NIPS Workshop on Machine Learning Systems (LearningSys)*.

[17] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proc. of USENIX OSDI*.

[18] Bulpitt Ci. 1987. Confidence Intervals. *Lancet* (1987).

[19] Jinkun Geng, Dan Li, and Shuai Wang. 2019. Accelerating Distributed Machine Learning by Smart Parameter Server. In *Proc. of APNet*.

[20] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. 2013. Speech Recognition with Deep Recurrent Neural Networks. In *Proc. of IEEE ICASSP*.

[21] Aaron Harlap, Henggang Cui, Wei Dai, Jinliang Wei, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, and Eric P Xing. 2016. Addressing the Straggler Problem for Iterative Convergent Parallel ML. In *Proc. of ACM SoCC*.

[22] Aaron Harlap, Alexey Tumanov, Andrew Chung, Gregory R Ganger, and Phillip B Gibbons. 2017. Proteus: Agile ML Elasticity through Tiered Reliability in Dynamic Resource Markets. In *Proc. of ACM EuroSys*.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proc. of IEEE CVPR*.

[24] Sergey Ioffe and Christian Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proc. of ICML*.

[25] Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. 2017. Heterogeneity-aware Distributed Parameter Servers. In *Proc. of ACM SIGMOD*.

[26] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters. In *Proc. of USENIX OSDI*.

[27] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proc. of NIPS*.

[28] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *Proc. of USENIX OSDI*.

[29] Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. 2018. Asynchronous Decentralized Parallel Stochastic Gradient Descent. In *Proc. of ICML*.

[30] Qinyi Luo, Jinkun Lin, Youwei Zhuo, and Xuehai Qian. 2019. Hop: Heterogeneity-aware Decentralized Training. In *Proc. of ACM ASPLOS*.

[31] Andrew Or, Haoyu Zhang, and Michael J Freedman. 2020. Resource Elasticity in Distributed Deep Learning. In *Proc. of Machine Learning and Systems (MLSys)*.

[32] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic Differentiation in PyTorch. In *Proc. of NIPS Autodiff Workshop*.

[33] Pitch Patarasuk and Xin Yuan. 2007. Bandwidth Efficient All-reduce Operation on Tree Topologies. In *Proc. of IEEE IPDPS*.

[34] Pitch Patarasuk and Xin Yuan. 2009. Bandwidth Optimal All-reduce Algorithms for Clusters of Workstations. *J. Parallel and Distrib. Comput.* (2009).

[35] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters. In *Proc. of ACM EuroSys*.

[36] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, Chen Meng, and Wei Lin. 2019. DL2: A Deep Learning-driven Scheduler for Deep Learning Clusters. *arXiv preprint arXiv:1909.06040* (2019).

[37] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A Generic Communication Scheduler for Distributed DNN Training Acceleration. In *Proc. of ACM SOSP*.

[38] Aurick Qiao, Abutalib Aghayev, Weiren Yu, Haoyang Chen, Qirong Ho, Garth A Gibson, and Eric P Xing. 2018. Litz: Elastic Framework for High-Performance Distributed Machine Learning. In *Proc. of USENIX ATC*.

[39] Herbert Robbins and Sutton Monro. 1951. A Stochastic Approximation Method. *The Annals of Mathematical Statistics* (1951).

[40] Shriram Sarvotham, Rudolf Riedi, and Richard Baraniuk. 2001. Connection-Level Analysis and Modeling of Network Traffic. In *Proc. of ACM SIGCOMM Workshop on Internet Measurement*.

[41] Alexander Sergeev and Mike Del Balso. 2018. Horovod: Fast and Easy Distributed Deep Learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[42] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-scale Image Recognition. In *Proc. of ICLR*.

[43] Peng Sun, Yonggang Wen, Nguyen Binh Duong Ta, and Shengen Yan. 2017. Towards Distributed Machine Learning in Shared Clusters: A Dynamically-Partitioned Approach. In *Proc. of IEEE International Conference on Smart Computing*.

[44] Richard S Sutton and Andrew G Barto. 1998. *Reinforcement Learning: An Introduction.* MIT Press Cambridge.

[45] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going Deeper with Convolutions. In *Proc. of IEEE CVPR*.

[46] Rashish Tandon, Qi Lei, Alexandros G Dimakis, and Nikos Karampatziakis. 2017. Gradient Coding: Avoiding Stragglers in Distributed Learning. In *Proc. of ICML*.

[47] Eric P Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. 2015. Petuum: A New Platform for Distributed Machine Learning on Big Data. *IEEE Transactions on Big Data* (2015).

[48] Xing Zhao, Aijun An, Junfeng Liu, and Bao Xin Chen. 2019. Dynamic Stale Synchronous Parallel Distributed Training for Deep Learning. In *Proc. of IEEE ICDCS*.