Distributed Machine Learning through Heterogeneous Edge Systems

Hanpeng Hu,[1] **Dan Wang,**[2] **Chuan Wu**[1]
[1]The University of Hong Kong, [2]The Hong Kong Polytechnic University
hphu@cs.hku.hk, csdwang@comp.polyu.edu.hk, cwu@cs.hku.hk.

Abstract

Many emerging AI applications request distributed machine learning (ML) among edge systems (e.g., IoT devices and PCs at the edge of the Internet), where data cannot be uploaded to a central venue for model training, due to their large volumes and/or security/privacy concerns. Edge devices are intrinsically *heterogeneous* in computing capacity, posing significant challenges to parameter synchronization for parallel training with the parameter server (PS) architecture. This paper proposes *ADSP*, a parameter synchronization model for distributed machine learning (ML) with heterogeneous edge systems. Eliminating the significant waiting time occurring with existing parameter synchronization models, the core idea of *ADSP* is to let faster edge devices continue training, while committing their model updates at strategically decided intervals. We design algorithms that decide time points for each worker to commit its model update, and ensure not only global model convergence but also faster convergence. Our testbed implementation and experiments show that *ADSP* outperforms existing parameter synchronization models significantly in terms of ML model convergence time, scalability and adaptability to large heterogeneity.

1 Introduction

Many edge-based AI applications have emerged in recent years, where various edge systems (e.g., PCs, smart phones, IoT devices) collect local data, collaboratively train a ML model, and use the model for AI-driven services. For example, smart cameras are deployed in surveillance systems (Ved 2019; Park et al. 2018), which capture local images/videos and train a global face recognition model aggregately. In Industry AI Operations (AIOps) (Qu and Ha 2017), chillers in a building or an area collect temperature and electricity consumption data in the households, and derive a global COP (Coefficient of Performance) prediction model (Chen et al. 2019).

A straightforward way of training a global model with data collected from multiple edge systems is to send all data to a central venue, e.g., a cloud data center, and train the datasets using a ML framework, such as TensorFlow (Abadi et al. 2016), MXNet (Chen et al. 2015) and Caffe2 (Hazelwood et al. 2018)). Such a 'data aggregation $\rightarrow$ training' approach may well incur large network bandwidth cost, due to large data transmission volumes and continuous generation nature of the data flow, as well as data security and privacy concerns. To alleviate these issues, collaborative, distributed training among edge systems has been advocated (Tao and Li 2018), where each edge system locally trains the dataset it collects, and exchanges model parameter *updates* (i.e., gradients) with each other through parameter servers (Wang et al. 2018; Konečný et al. 2016; Park et al. 2018) (a.k.a., geo-distributed data parallel training).

Edge systems are intrinsically heterogeneous: their hardware configurations can be vastly different, leading to different computation and communication capacities. This brings significant new issues on parameter synchronization among the edge workers. In a data center environment, synchronous training (i.e., Bulk Synchronous Parallel (BSP) (Ho et al. 2013; Li et al. 2013; Low et al. 2012)) is adopted by the majority of production ML jobs (based on our exchanges with large AI cloud operators), given the largely homogeneous worker configuration: each worker trains a mini-batch of input data and commits computed gradients to the PS; the PS updates global model after receiving commits from all workers, and then dispatches updated model parameters to all workers, before each worker can continue training its next mini-batch. In the edge environment, the vastly different training speeds among edge devices call for a more asynchronous parameter synchronization model, to expedite ML model convergence.

Stale Synchronous Parallel (SSP) (Ho et al. 2013) and *Totally Asynchronous Parallel (TAP)* (Hsieh et al. 2017) are representative asynchronous synchronization models. With TAP, the PS updates the global model upon commit from each individual worker, and dispatches updated model immediately to the respective worker; it has been proven that such complete asynchrony cannot ensure model convergence (Hsieh et al. 2017). SSP enforces bounded asynchronization: fast workers wait for slow workers for a bounded difference in their training progress, in order to ensure model convergence. A few recent approaches have been proposed to further improve convergence speed of asynchronous training(Hadjis et al. 2016; Wang and Joshi 2018) (see more in Sec. 2).

We investigate how existing parameter synchronization models work in a heterogeneous edge environment with testbed experiments (Sec. 2.3), and show that the waiting time (overall model training time minus gradient computation time) is still more than 50% of the total training time with the representative synchronization models.

Aiming at minimizing the waiting time and optimizing computing resource utilization, we propose *ADSP* (ADaptive Synchronous Parallel), a new parameter synchronization model for distributed ML with heterogeneous edge systems. Our core idea is to let faster workers continue with their mini-batch training all the time, while enabling all workers to commit their model updates at the same strategically decided intervals, to ensure not only model convergence but also faster convergence. The highlights of *ADSP* are summarized as follows:

▷ *ADAP* is tailored for distributed training in heterogeneous edge systems, which fully exploits individual workers' processing capacities by eliminating the waiting time.

▷ *ADSP* actively controls the parameter update rate from each worker to the PS, to ensure that the total number of commits from each worker to the PS is roughly the same over time, no matter how fast or slow each worker performs local model training. Our algorithm exploits a momentum-based online search approach to identify the best cumulative commit number across all workers, and computes the commit rates of individual workers accordingly. ADSP is proven to converge after a sufficient number of training iterations.

▷ We have done a full-fledged implementation of *ADSP* and evaluated it with real-world edge ML applications. Evaluation results show that it outperforms representative parameter synchronization models significantly in terms of model convergence time, scalability and adaptability to large heterogeneity.

2 Background and Motivation

2.1 SGD in PS Architecture

Stochastic Gradient Descent (SGD) is the widely used algorithm for training neural networks (Hadjis et al. 2016; Abadi et al. 2016). Let W_t be the set of global parameters of the ML model at t. A common model update method with SGD is:

$$W_{t+1} = W_t - \eta \nabla \ell(W_t) + \mu(W_t - W_{t-1}) \tag{1}$$

where $\nabla \ell(W_t)$ is the gradient, η is the learning rate, and $\mu \in [0, 1]$ is the *momentum* introduced to accelerate the training process, since it accumulates gradients in the right direction to the optimal point (Polyak 1964; Sutskever et al. 2013).

In widely adopted data-parallel training with the parameter server (PS) architecture(Chilimbi et al. 2014), SGD update rule can be applied at both the workers and the PS (Jiang et al. 2017). Each worker holds a local copy of the ML model, its local dataset is divided into mini-batches, and the worker trains its model in an iterative fashion: in each step, the worker calculates gradients of model parameters using one mini-batch of its data, and may commit its gradients to the PS and pull the newest global model parameters from the PS. The PS updates the global model using Eqn. (1) with gradients received from the workers and a *global learning rate* η. In the case that a worker does not synchronize model parameters with the PS per step, the worker may carry out local model updates using computed gradients according to Eqn. (1), where the gradients are multiplied by a *local learning rate* η'.

2.2 Existing Parameter Synchronization Models

A parameter synchronization model specifies when each worker commits its gradients to the PS and whether it should be synchronized with updates from other workers; it critically affects convergence speed of model training. Three representative synchronization models, BSP, SSP and TAP, have been compared in (Hsieh et al. 2017), which proves that BSP and SSP guarantee model convergence whereas TAP does not. Training convergence with BSP is significantly slower than SSP (Ho et al. 2013), due to BSP's strict synchronization barriers. Based on the three synchronization models, many studies have followed, aiming to reduce the convergence time by reducing communication contention or overhead (Chen, Wang, and Li 2019; Lin et al. 2017; Sun et al. 2016), adjusting the learning rate (Jiang et al. 2017), and others (Zhang et al. 2018). ADACOMM (Wang and Joshi 2018) allows accumulating local updates before committing to the PS, and adopts BSP-style synchronization model, i.e., all workers run τ training steps before synchronizing with the PS. It also suggests reducing the commit rate periodically according to model loss; however, the instability in loss values during training and the rapidly declining commit rate are not ideal for expediting training (according to our experiments).

Aiming at minimizing waiting time among heterogeneous workers, our synchronization model, ADSP, employs an online search algorithm to automatically find the optimal/near-optimal update commit rate for each worker to adopt.

2.3 Impact of Waiting

We divide the time a worker spends in each training step into two parts: (i) the *computation time*, to carry out forward propagation to produce output and backward propagation to compute gradients/apply model updates (Chen et al. 2015); and (ii) the *waiting time*, including the time for exchanging gradients/parameters with the PS and the blocked time due to synchronization barrier (i.e., the time when the worker is not doing computation nor communication).

We experiment with representative synchronization models to investigate their waiting time incurred. We train a convolutional neural network (CNN) model on the Cifar10 dataset (Krizhevsky and Hinton 2010) with 1 PS and 3 workers with heterogeneous computation capacities (time ratio to train one mini-batch is 1:1:3). Fig. 1 shows the convergence time (overall training time to model convergence) and the average time spent per training step, incurred with BSP, SSP, and ADACOMM (See Sec. 5 for their details). TAP is not compared as it has no convergence guarantee. The computation/waiting time is averaged among all workers. We

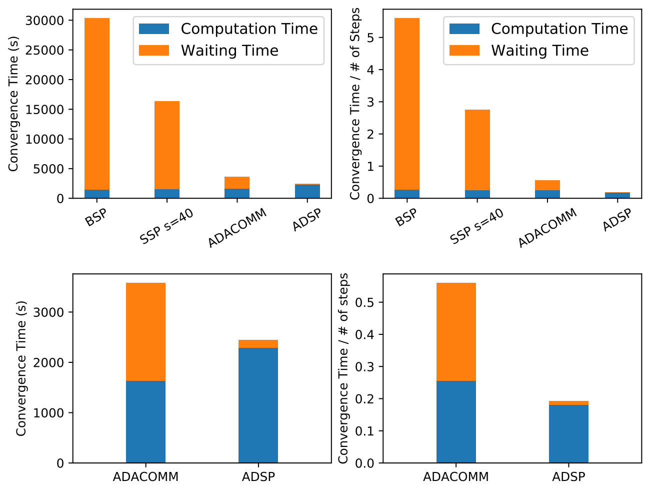

Figure 1: Training time breakdown with different parameter synchronization models.

see that with heterogeneous workers, the waiting time dominates the overall training time with BSP and SSP, and their overall convergence time and time spent per training step are long. With ADACOMM, the waiting time and overall training time are much shorter. Nevertheless, its waiting time is still close to half of the total training time, i.e., the effective time used for model training is only around 50%, due to its relative conservative approach on local model updates.

Our key question is: what is the limit that we can further reduce the waiting time to, such that time is spent most efficiently on model computation and convergence can be achieved in the most expedited fashion? Our answer, ADSP, allows fast workers to keep training while maintaining approximately the same gradient commit rates among all workers. Fig. 1 shows that the waiting time is minimized to a negligible level with ADSP, as compared to the computation time. As such, almost all training time is effectively used for model computation and fast model convergence is achieved.

3 ADSP Overview

We consider a set of heterogeneous edge systems and a parameter server (PS) located in a datacenter, which together carry out SGD-based distributed training to learning a ML model. ADSP (ADaptive Synchronous Parallel) is a new parameter synchronization model for this distributed ML system. The design of ADSP targets the following goals: (i) make full use of the computation capacity of each worker; (ii) choose a proper commit rate to balance the tradeoff between hardware efficiency (utilization of worker computing capacity) and statistical efficiency (i.e., reduction of loss per training step), in order to minimize the overall time taken to achieve model convergence; (iii) ensure model convergence under various training speeds and bandwidth situations at different workers.

With ADSP, time is divided into equal-sized slots of duration $\Gamma > 0$: $0, \Gamma, 2\Gamma, \ldots$, which we call as *check periods*. We refer to time points $\Gamma, 2\Gamma, \ldots, p\Gamma, \ldots$, as *checkpoints*.

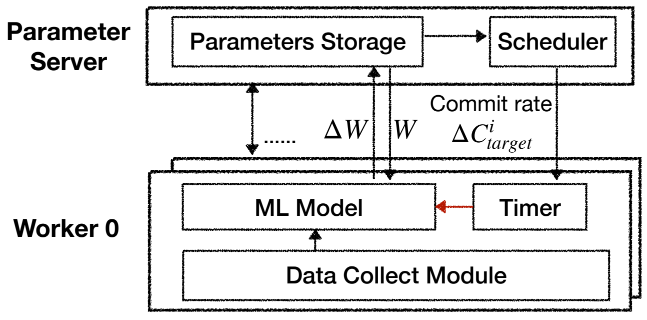

Figure 2: ADSP workflow.

More precisely, we define the process of a worker sending computed gradients to the PS as a *commit*, and the number of commits from worker i during a check period as *commit rate* ΔC^i_{target}. ADSP consists of two modules: 1) a novel synchronization model, which allows faster edge systems to perform more training before each update to the PS, and ensures that the commit rates of all worker are the same; 2) a global commit rate search algorithm, which selects an appropriate commit rate for all workers to pursue, in order to achieve fast convergence.

Let c_i denote the total number of commits from worker i to the PS, since the very beginning. At each checkpoint, we compute the target total number of commits that each worker is expected to have submitted by the next checkpoint, C_{target}, and adjust the commit rate of each worker i in the next check period as $\Delta C^i_{target} = C_{target} - c_i$, respectively.

Fig. 2 shows the workflow of our ADSP model. The data produced/collected at each edge system/worker is stored into training datasets. For each mini-batch in its dataset, an edge system computes a *local update* of model parameters, i.e., gradients, using examples in this mini-batch. After training one mini-batch, it moves on to train the next mini-batch and derives another local update. Worker i pushes its *accumulative update* (i.e., sum of all gradients it has produced since last commit multiplied with the local learning rate) according to the commit rate ΔC^i_{target}. A scheduler adjusts and informs each end system of the target commit rate ΔC^i_{target} over time. Upon receiving a commit from worker i, the PS multiplies the accumulated update with the *global learning rate* (Jiang et al. 2017) and then updates the global model with it; worker i then pulls updated parameters from the PS and continues training the next mini-batch.

4 ADSP Algorithms and Analysis

It is common to have large heterogeneity among edge systems, including different computation power and network delays to the datacenter hosting the PS. Our core idea in designing ADSP is to *adapt* to the heterogeneity, i.e., to transform the training in heterogeneous settings into homogeneous settings using a *no-waiting* strategy: *we allow different workers to process different numbers of mini-batches between two commits according to their training speed, while ensuring the number of commits of all the workers approxi-*

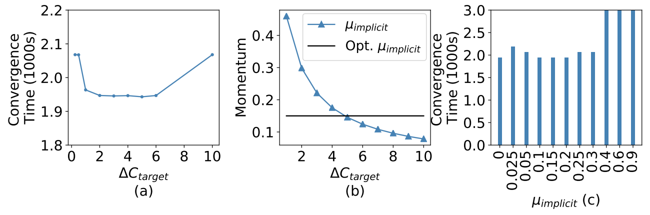

Figure 3: (a) the impact of C_{target} on convergence time; (b) an illustration of $\mu_{implicit}$; (c) convergence time with different $\mu_{implicit}$ values.

mately equal at periodical checkpoints. To achieve this, we mainly control the hyper-parameter, *commit rate*, making faster workers accumulate more local updates before committing their updates, so as to eliminate the waiting time. By enforcing approximately equal numbers of commits from all the workers over time, we can ensure model convergence.

4.1 The Impact of C_{target} on Convergence

The target total number of commits to be achieved by each worker by the next checkpoint, C_{target}, decides commit rate of each worker i within the next check period, as $\Delta C^i_{target} = C_{target} - c_i$ (c_i is i's current total commit number). The commit rate has a significant impact on the training progress: if ΔC^i_{target} is large, a slow worker may fail to achieve that many commits in the next period, due to the limited compute capacity; even if it can hit the target, too many commits may incur high communication overhead, which in turn slows down the training process. On the other hand, if the number of target commits is too small, which implies that each end system commits its gradients after many steps of mini-batch training using local parameters, large difference and significant staleness exist among the model copies at different workers, which may adversely influence model convergence as well.

To illustrate this, we train a CNN model on the Cifar10 dataset (Krizhevsky and Hinton 2010) with 1 PS and 3 workers (time ratio to train one mini-batch is 1:1:3), where all workers keep training their mini-batches and commit gradients to the PS at the same commit rate ΔC_{target} over time. We vary the value of ΔC_{target} in different runs of the experiment. Fig. 3(a) shows that with the increase of ΔC_{target}, the model convergence time becomes smaller at first and then increases. This is consistent with our discussions above.

We next quantify the effect of the commit rate ΔC_{target} on model convergence. Suppose that all m workers communicate with the PS independently. Let $U(W_t)$ denote the accumulative local updates that a worker commits when the global model is W_t, and v_i denote the number of steps that worker i can train per unit time. We have the following theorem.

Theorem 1. *Set the momentum μ in the SGD update formula (1) to zero. The expected SGD update on the global model is equivalent to*

$$\mathbb{E}(W_{t+1} - W_t) = (1-p)\mathbb{E}(W_t - W_{t-1}) - p\eta\mathbb{E}U(W_t) \quad (2)$$

$$\textit{where } \; p = 1/(1 + (1-1/m)\sum_{i=1}^{m} \frac{\Gamma}{\Delta C^i_{target} v_i}) \quad (3)$$

The detailed proof is given in the supplemental file (Hu, Wang, and Wu 2019). Compared to the SGD update formula in Eqn. (1), the result is interesting: with our ADSP model, staleness induced by cumulative local updates can be considered as inducing an extra momentum term (i.e., $1-p$) into the SGD update equation. To distinguish this term from the original momentum μ in Eqn. (1), we refer to this term as the *implicit momentum*, denoted by $\mu_{implicit} = 1-p$. As we increase ΔC^i_{target}, the implicit momentum becomes smaller according to Eqn. (2).

With the same CNN training experiments as above, Fig. 3(b) illustrates how $1-p$ varies with ΔC_{target} (according to Eqn. (3)). The optimal momentum is derived based on Fig. 3(c), where we vary the value of $\mu_{implicit}$ in Eqn. (2) in our experiments, and show how the time taken for model convergence varies with different $\mu_{implicit}$ values. Inspired by the observations, we seek to identify the best commit rate ΔC_{target} for the workers, that decides the best $\mu_{implicit}$ to achieve the shortest convergence time.

4.2 The Commit Rate Search Algorithm

We propose a local search method to identify a near-optimal commit rate to achieve the fastest convergence, exploiting the observations that the staleness induced by local updates can be converted to an implicit momentum term in SGD update and the implicit momentum decreases as we increase the commit rate. The algorithm is given in Alg. 1, which is executed by the scheduler (Fig. 2).

In the algorithm, an *epoch* is a time interval containing multiple check periods, for commit rate adjustment. At the beginning of each *epoch* (e.g., 1 hour), the scheduler performs the search for the optimal commit rates of workers in this epoch. We start with a small target total commit number C_{target}, allowing each worker to commit at least once in each check period; in this case, the commit rates ΔC^i_{target}'s are small, asynchrony-induced implicit momentum is large, and the corresponding point in Fig. 3(b) is located to the left of the optimal momentum. Then the scheduler evaluates the training performance (i.e., loss decrease speed, to be detailed in Sec. 4.2) induced by C_{target} and $C_{target}+1$, by running the system using commit rates computed based on the two values for a specific period of time (e.g., 1 minute). If $C_{target}+1$ leads to better performance, the scheduler repeats the search, comparing performance achieved by $C_{target}+1$ and $C_{target}+2$ further; otherwise, the search stops and the commit rates ΔC^i_{target}'s decided by the current C_{target} are used for the rest of this epoch. The rationale behind is that the optimal C_{target} for each epoch is larger than the initial value ($\max_{i=1,\ldots,M} c_i + 1$), so we only need to determine whether to increase it or not.

Online Search and Reward Design. Traditional search methods are usually offline (Hadjis et al. 2016), blocking the

Algorithm 1 Commit Rate Adjustment at the Scheduler

```
1:  function MAINFUNCTION
2:      for epoch e = 1, 2, ... do
3:          $C_{target} = \max_{i=1,...,M} c_i + 1$
4:          $C_{target} \leftarrow$ DECIDECOMMITRATE($C_{target}$)
5:          run PARAMETERSERVER and WOKERS for the remaining time
6:      end for
7:  end function
8:  function DECIDECOMMITRATE($C_{target}$)
9:      $r_1 \leftarrow$ ONLINEEVALUATE($C_{target}$)
10:     $r_2 \leftarrow$ ONLINEEVALUATE($C_{target} + 1$)
11:     if $r_2 > r_1$ then
12:         return DECIDECOMMITRATE($C_{target} + 1$).
13:     else
14:         return $C_{target}$
15:     end if
16: end function
17: function ONLINEEVALUATE($C_{target}$)
18:     for $i$ = 0, 1, 2, ...,M do
19:         $\Delta C^i_{target} = C_{target} - c_i$
20:         Send $\Delta C^i_{target}$ to worker $i$
21:     end for
22:     Training for 1 minute
23:     return reward $r$
24: end function
```

whole system when trying out a specific set of variable values and trying each configuration starting with the same system state. With an offline search method, one can select the best configuration by comparing the final loss achieved after running different configurations for the same length of time. However, such a search process incurs significant extra delay into the training progress and hence significant slowdown of model convergence. In Alg. 1, we instead adopt an *online search* method (in DECIDECOMMITRATE()): we consecutively run each configuration for a specific time (e.g., 1 minute) without blocking the training process.

To compare the performance of the configurations when they do not start with the same system state, we define a *reward* as follows. The loss convergence curve of SGD training usually follows the form of $O(1/t)$ (Peng et al. 2018). We collect a few (time t, loss ℓ) pairs when the system is running with a particular configuration, e.g., at the start, middle and end of the 1 minute period, and use them to fit the following formula on the left:

$$\ell = \frac{1}{a_1^2 t + a_2} + a_3 \quad \Rightarrow \quad r = \frac{a_1^2}{\frac{1}{\ell - a_3} - a_2}$$

where a_1, a_2, a_3 are parameters. Then we obtain the reward r as the loss decrease speed, by setting ℓ to a constant and calculating the reciprocal of corresponding t. The target of the online search algorithm is to find the commit rate that reaches the maximum reward, i.e., the minimum time to converge to a certain loss.

Algorithm 2 ADSP: Worker and PS Procedures

```
End System:  i = 1, 2, ..., m
1:  function WORKER
2:      for epoch $e = 1, 2, \ldots$ do
3:          receive $\Delta C^i_{target}$ from the scheduler
4:          set a timer with a timeout of $\frac{\Gamma}{\Delta C^i_{target}} - \mathcal{O}_i$ and invoking TimeOut() upon timeout
5:          while model not converged do
6:              train a minibatch to obtain gradient $g_i$
7:              accumulated gradient $U_i \leftarrow U_i + \eta' g_i$ ($\eta'$ is the local learning rate)
8:          end while
9:      end for
10: end function
11: function TIMEOUT
12:     commit $U_i$ to the PS
13:     receive updated global model parameters from the PS and update local model accordingly
14:     restart the timer with timeout of $\frac{\Gamma}{\Delta C^i_{target}} - \mathcal{O}_i$
15: end function
Parameter Server:
1:  function PARAMETERSERVER
2:      while model not converged do
3:          if receive commit $U_i$ from worker $i$ then
4:              $W \leftarrow W - \eta U_i$
5:              Send $W$ to worker $i$
6:          end if
7:      end while
8:  end function
```

4.3 Worker and PS Procedures

The procedures at each end system (i.e., worker) and the PS with ADSP is summarized in Alg. 2, where $\mathcal{O}_i$ represents the communication time for worker i to commit an update to the PS and pull the updated parameters back. At each worker, we use a timer to trigger commit of local accumulative model update to the PS asynchronously, once every $\frac{\Gamma}{\Delta C^i_{target}} - \mathcal{O}_i$ time.

4.4 Convergence Analysis

We show that ADSP in Alg. 2 ensures model convergence. We define $f_t(W)$ as the objective loss function at step t with global parameter state W, where t is the global number of steps (i.e., cumulative number of training steps carried out by all workers). Let $\tilde{W}_t$ be the set of global parameters obtained by ADSP right after step t, and W^* denote the optimal model parameters that minimize the loss function. We make the following assumptions on the loss function and the learning rate, which are needed for our convergence proof, but are not followed in our experimental settings.

Assumptions:

(1) $f_t(W)$ is convex

(2) $f_t(W)$ is L-Lipschitz, i.e., $\|\nabla f_t\| \leqslant L$

(3) The learning rate decreases as $\eta_t = \frac{\eta}{\sqrt{t}}$, $t = 1, 2, \ldots$, where η is a constant.

Based on the assumptions, we have the following theorem on training convergence of ADSP.

Theorem 2 (Convergence)**.** *ADSP ensures that by each checkpoint, the numbers of update commits submitted by any two different workers i_1 and i_2 are roughly equal, i.e., $c_{i_1} \approx c_{i_2}$. The regret $R = \sum_{t=1}^{T} f_t(\tilde{W}_t) - f(W^*)$ is upper-bounded by $O(\sqrt{T})$, when $T \to +\infty$.*

The *regret* is the accumulative difference between the loss achieved by ADSP and the optimal loss over the training course. When the accumulative difference is under a sub-linear bound about T (where T is the total number of parameter update steps at the PS), we have $f_t(\tilde{W}_t) \to f(W^*)$ when t is large. Then $R/T \to 0$ as $T \to +\infty$, showing that our ADSP model converges to the optimal loss. The detailed proof is given in the supplemental file (Hu, Wang, and Wu 2019).

5 Performance Evaluation

We implement ADSP as a ready-to-use Python library based on TensorFlow (Abadi et al. 2016), and evaluate its performance with testbed experiments.

5.1 Experiment Setup

Testbed. We emulate heterogeneous edge systems following the distribution of hardware configurations of edge devices in a survey (Jkielty 2019), using 19 Amazon EC2 instances (Wang and Ng 2010): 7 `t2.large` instances, 5 `t2.xlarge` instances, 4 `t2.2xlarge` instances and 2 `t3.xlarge` instances as workers, and 1 `t3.2xlarge` instance as the PS.

Applications. We evaluate ADSP with three distributed ML applications: (i) image classification on Cifar-10 (Krizhevsky and Hinton 2010) using a CNN model from the TensorFlow tutorial (Tensorflow 2019); (ii) Fatigue life prediction of bogies on high-speed trains, training a recurrent neural network (RNN) model with the dataset collected from the China high-speed rail system; (iii) Coefficient of Performance (COP) prediction of chillers, training a global linear SVM model with a chiller dataset.

Baselines. (1) *SSP* (Ho et al. 2013), which allows the fastest worker to run ahead of the slowest worker by up to s steps; (2) *BSP* (Valiant 1990), where the PS strictly synchronizes among all workers such that they always perform the same number of training steps. (3) *ADACOMM* (Wang and Joshi 2018), which allows all workers to accumulate τ updates before synchronizing with the PS and reduces τ periodically. (4) *Fixed ADACOMM*, a variant of ADACOMM with τ fixed for all workers.

Default Settings. By default, each mini-batch in our model training includes 128 examples. The check period of ADSP is 60 seconds, and each epoch is 20 minutes long. The global learning rate is $1/M$ (which we find works well through experiments). The local learning rate is initialized to 0.1 and decays exponentially over time.

5.2 Experiment Results

All results given in the following are based on CNN training on the Cifar-10 dataset. More experiment results on fatigue life prediction and CoP prediction are given in the supplemental file (Hu, Wang, and Wu 2019).

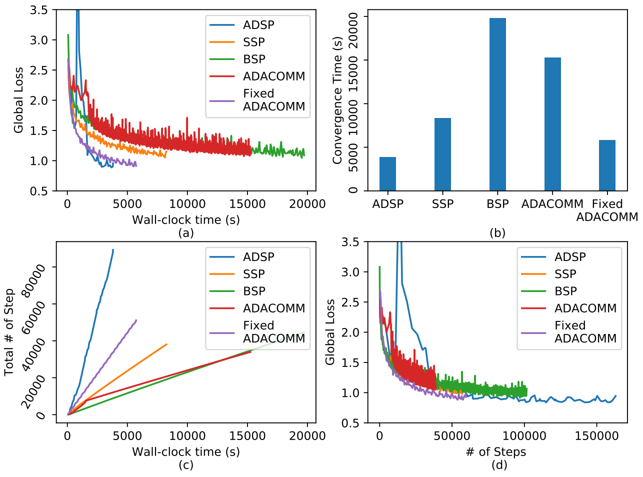

Figure 4: Comparison of ADSP with baselines in training efficiency: training CNN on the Cifar-10 dataset.

Performance of ADSP. We compare ADSP with the baselines in terms of the wall-clock time and the number of training steps needed to reach model convergence, to validate the effectiveness of no-waiting training of ADSP. In Fig. 4, the global loss is the loss evaluated on the global model on the PS, and the number of steps is the cumulative number of steps trained at all workers. We stop training, i.e., decide that the model has converged, when the loss variance is smaller than a small enough value for 10 steps. Fig. 4(a) plots the loss curves and Fig. 4(b) correspondingly shows the convergence time with each method. We see that ADSP achieves the fastest convergence: 80% acceleration as compared to BSP, 53% to SSP, and 33% to Fixed ADACOMM. For ADACOMM, although we have used the optimal hyper-parameters as in (Wang and Joshi 2018), it converges quite slowly, which could be due to its instability in tuning τ: τ is tuned periodically based on the current loss; if the loss does not decrease, it simply multiplies τ with a constant. In Fig. 4(c), we see that ADSP carries out many more training steps within its short convergence time, which may potentially lead to a concern on its training efficiency. Fig. 4(d) further reveals that the per-training-step loss decrease achieved by ADSP is slightly lower than that of Fixed ADACOMM, and better than other baselines. The spike in ADSP curve at the beginning stage is due to small commit rates that our search algorithm derives, which make the loss fluctuates significantly. However, with ADSP, the model eventually converges to a smaller loss than losses that other baselines converge to.

Adaptability to Heterogeneity. We next evaluate ADSP's adaptability to different levels of end system heterogeneity. Besides hardware configuration difference among the workers, we further enable each worker to sleep for a specific short time after each step of training one mini-batch, and

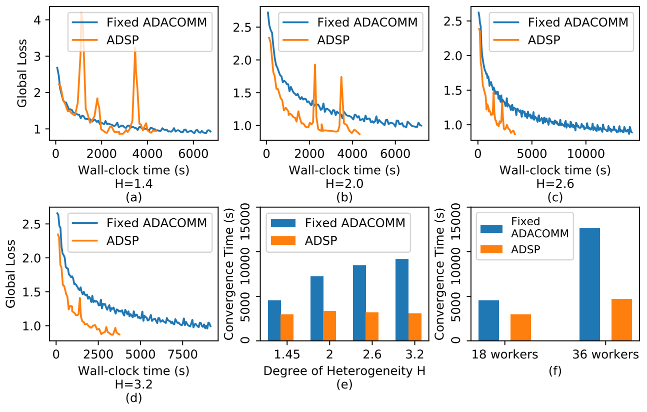
Figure 5: Comparison of ADSP with Fixed ADACOMM at different degrees of heterogeneity and system scales.

tune the sleep time to adjust training speeds of workers. We define the heterogeneity degree among the workers as follows:

$$H = \frac{\sum_i^M v_i/M}{\min_{i=1,\dots,M} v_i}$$

where v_i is the number of mini-batches that worker i can process per unit time. The discussion of the heterogeneity degree considering communication overhead is given in our supplemental file (Hu, Wang, and Wu 2019).

Since BSP, SSP and ADACOMM are significantly slower than ADSP in training convergence, here we only compare ADSP with Fixed ADACOMM. Fig. 5(a)-(d) show that ADSP achieves faster convergence than Fixed ADACOMM (though with more spikes) in different heterogeneity levels. The corresponding convergence times are summarized in Fig. 5(e), which shows that the gap between ADSP and Fixed ADACOMM becomes larger when the workers differ more in training speeds. ADSP achieves a 62.4% convergence speedup as compared to Fiexd ADACOMM when $H = 3.2$. The reason lies in that Fixed ADACOMM still enforces faster workers to stop and wait for the slower workers to finish τ local updates, so the convergence is significantly influenced by the slowest worker. With ADSP, the heterogeneity degree hardly affects the convergence time much, due to its no-waiting strategy. Therefore, ADSP can adapt well to heterogeneity in end systems.

System Scalability We further evaluate ADSP with 36 workers used for model training, whose hardware configuration follows the same distribution as in the 18-worker case. Fig. 5(f) shows that when the worker number is larger, both ADACOMM and ADSP become slower, and ADSP still achieves convergence faster than Fixed ADACOMM (which is more obvious than in the case of smaller worker number). Intuitively, when the scale of the system becomes larger, the chances increase for workers to wait for slower ones to catch up, resulting in that more time being wasted with Fixed ADACOMM; ADSP can use this part of time to do more training, and is hence a more scalable solution in big ML training jobs.

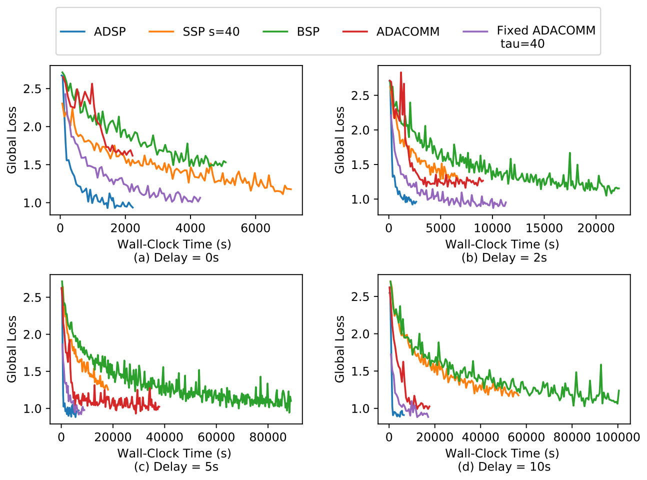
Figure 6: Comparison of ADSP with baselines with different network delays.

The Impact of Network Latency. Edge systems usually have relatively poor network connectivity (Konečný et al. 2016); the communication time for each commit is not negligible, and could be even larger than the processing time in each step. Fig. 6 presents the convergence curve of each method as we add different extra delays to the communication module. When we increase the communication delay, the speed-up ratio of *ADSP*, *Adacomm* and *Fixed Adacomm*, as compared to *BSP* and *SSP*, becomes larger. This is because the first three models allow local updates and commit to the PS less frequently, consequently less affected by the communication delay than the last two methods. Among the first three models, ADSP still performs the best in terms of convergence speed, regardless of the communication delay.

The rationale behind is that we can count the communication time when evaluating a worker's '*processing capacity*': for worker i, the average processing time per training step is $t_i + \mathcal{O}_i/\tau_i$, where t_i is the time to train a mini-batch, $\mathcal{O}_i$ is the communication time for each commit, and τ_i is the number of local updates between two commits.Therefore, we can extend the scope of heterogeneity in processing capacity to include the heterogeneity of communication time as well. ADSP only needs to ensure the commit rates of all workers are consistent, and can inherently handle the generalized heterogeneity without regard to which components cause the heterogeneity.

6 Concluding Remarks

This paper presents ADSP, a new parameter synchronization model for distributed ML with heterogeneous edge systems. ADSP allows workers to keep training with minimum waiting and enforces approximately equal numbers of commits from all workers to ensure training convergence. An online search algorithm is carefully devised to identify the near-optimal global commit rate. ADSP maximally exploits computation resources at heterogeneous workers, targeting training convergence in the most expedited fashion. Our testbed experiments show that ADSP achieves up to 62.4% convergence acceleration as compared to most of the state-of-the-art parameter synchronization models. ADSP is also well adapted to different degrees of heterogeneity and large-scale ML applications.

7 Acknowledgements

This work was supported in part by grants from Hong Kong RGC GRF HKU 17204715, 17225516, PolyU 15210119, CRF C7036-15G, C5026-18G, WHU-Xiaomi AI Lab, ITF UIM/363, PolyU 1-ZVPZ, and a Huawei Collaborative Grant.

References

Abadi, M.; Barham, P.; Chen, J.; Chen, Z.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Irving, G.; Isard, M.; et al. 2016. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, 265–283.

Chen, T.; Li, M.; Li, Y.; Lin, M.; Wang, N.; Wang, M.; Xiao, T.; Xu, B.; Zhang, C.; and Zhang, Z. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv:1512.01274*.

Chen, Q.; Zheng, Z.; Hu, C.; Wang, D.; and Liu, F. 2019. Data-driven task allocation for multi-task transfer learning on the edge. In *IEEE ICDCS*.

Chen, C.; Wang, W.; and Li, B. 2019. Round-robin synchronization: Mitigating communication bottlenecks in parameter servers. In *IEEE INFOCOM*, 532–540.

Chilimbi, T. M.; Suzue, Y.; Apacible, J.; and Kalyanaraman, K. 2014. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, volume 14, 571–582.

Hadjis, S.; Zhang, C.; Mitliagkas, I.; Iter, D.; and Ré, C. 2016. Omnivore: An optimizer for multi-device deep learning on cpus and gpus. *arXiv:1606.04487*.

Hazelwood, K.; Bird, S.; Brooks, D.; Chintala, S.; Diril, U.; Dzhulgakov, D.; Fawzy, M.; Jia, B.; Jia, Y.; Kalro, A.; et al. 2018. Applied machine learning at facebook: A datacenter infrastructure perspective. In *HPCA*, 620–629.

Ho, Q.; Cipar, J.; Cui, H.; Lee, S.; Kim, J. K.; Gibbons, P. B.; Gibson, G. A.; Ganger, G.; and Xing, E. P. 2013. More effective distributed ml via a stale synchronous parallel parameter server. In *NIPS*, 1223–1231.

Hsieh, K.; Harlap, A.; Vijaykumar, N.; Konomis, D.; Ganger, G. R.; Gibbons, P. B.; and Mutlu, O. 2017. Gaia: Geo-distributed machine learning approaching lan speeds. In *NSDI*, 629–647.

Hu, H.; Wang, D.; and Wu, C. 2019. Distributed machine learning through heterogeneous edge systems. *arXiv:1911.06949*.

Jiang, J.; Cui, B.; Zhang, C.; and Yu, L. 2017. Heterogeneity-aware distributed parameter servers. In *SIGMOD*, 463–478.

Jkielty. 2019. The most popular smartphones in 2018. https://deviceatlas.com/blog/most-popular-smartphones.

Konečný, J.; McMahan, H. B.; Yu, F. X.; Richtárik, P.; Suresh, A. T.; and Bacon, D. 2016. Federated learning: Strategies for improving communication efficiency. *arXiv:1610.05492*.

Krizhevsky, A., and Hinton, G. 2010. Convolutional deep belief networks on cifar-10. *Unpublished manuscript*.

Li, M.; Zhou, L.; Yang, Z.; Li, A.; Xia, F.; Andersen, D. G.; and Smola, A. 2013. Parameter server for distributed machine learning. In *NIPS*, volume 6, 2.

Lin, Y.; Han, S.; Mao, H.; Wang, Y.; and Dally, W. J. 2017. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv:1712.01887*.

Low, Y.; Bickson, D.; Gonzalez, J.; Guestrin, C.; Kyrola, A.; and Hellerstein, J. M. 2012. Distributed graphlab: a framework for machine learning and data mining in the cloud. *PVLDB* 5(8):716–727.

Park, J.; Samarakoon, S.; Bennis, M.; and Debbah, M. 2018. Wireless network intelligence at the edge. *arXiv:1812.02858*.

Peng, Y.; Bao, Y.; Chen, Y.; Wu, C.; and Guo, C. 2018. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *EuroSys*.

Polyak, B. T. 1964. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics* 4(5):1–17.

Qu, X., and Ha, J. 2017. Next generation of devops: Aiops in practice@ baidu. In *SREcon17*.

Sun, P.; Wen, Y.; Duong, T. N. B.; and Yan, S. 2016. Timed dataflow: Reducing communication overhead for distributed machine learning systems. In *ICPADS*, 1110–1117.

Sutskever, I.; Martens, J.; Dahl, G.; and Hinton, G. 2013. On the importance of initialization and momentum in deep learning. In *ICML*, 1139–1147.

Tao, Z., and Li, Q. 2018. esgd: Communication efficient distributed deep learning on the edge. In *HotEdge*.

Tensorflow. 2019. Use tensorflow to train a cnn on cifar-10. https://github.com/tensorflow/models/tree/master/tutorials/image/cifar10.

Valiant, L. G. 1990. A bridging model for parallel computation. *Commun. ACM* 33(8):103–111.

Ved, M. 2019. Artificial intelligence (ai) solutions on edge devices. https://medium.com/@mehulved1503/artificial-intelligence-ai-solutions-on-edge-devices-1cc08d411a7c.

Wang, J., and Joshi, G. 2018. Adaptive communication strategies to achieve the best error-runtime trade-off in local-update sgd. *arXiv:1810.08313*.

Wang, G., and Ng, T. E. 2010. The impact of virtualization on network performance of amazon ec2 data center. In *IEEE INFOCOM*, 1–9.

Wang, S.; Tuor, T.; Salonidis, T.; Leung, K. K.; Makaya, C.; He, T.; and Chan, K. 2018. Adaptive federated learning in resource constrained edge computing systems. *J-SAC* 8:9.

Zhang, C.; Tian, H.; Wang, W.; and Yan, F. 2018. Stay fresh: Speculative synchronization for fast distributed machine learning. In *ICDCS*, 99–109.