Stay Fresh: Speculative Synchronization for Fast Distributed Machine Learning

Chengliang Zhang†, Huangshi Tian†, Wei Wang†, Feng Yan‡
†Hong Kong University of Science and Technology
‡University of Nevada, Reno
{czhangbn, htianaa, weiwa}@cse.ust.hk, fyan@unr.edu

***Abstract*—Large machine learning models are typically trained in parallel and distributed environments. The model parameters are iteratively refined by multiple worker nodes in parallel, each processing a subset of the training data. In practice, the training is usually conducted in an *asynchronous parallel* manner, where workers can proceed to the next iteration before receiving the latest model parameters. While this maximizes the rate of updates, the price paid is compromised training quality as the computation is usually performed using stale model parameters. To address this problem, we propose a new scheme, termed *speculative synchronization*. Our scheme allows workers to speculate about the recent parameter updates from others on the fly, and if necessary, the workers *abort* the ongoing computation, pull fresher parameters, and *start over* to improve the quality of training. We design an effective heuristic algorithm to judiciously determine when to restart training iterations with fresher parameters by quantifying the gain and loss. We implement our scheme in MXNet—a popular machine learning framework—and demonstrate its effectiveness through cluster deployment atop Amazon EC2. Experimental results show that speculative synchronization achieves up to 3× speedup over the *asynchronous parallel* scheme in many machine learning applications, with little additional communication overhead.**

I. Introduction

Large-scale machine learning (ML) has demonstrated state-of-the-art performance in many practical applications, such as voice-driven personal assistants [1], photo search [2] and captioning [3], and autonomous vehicles [4]. However, training large ML models at scale requires processing a massive amount of data, which cannot be accommodated by a single machine.

Distributed ML systems come as a solution that enables parallel training of large ML models in a cluster of machines [5]–[9]. In distributed ML systems, the training data is partitioned across many *worker* nodes. All workers share access to the model parameters, sharded across multiple servers. Each worker iteratively refines the parameters based on its subset of training data, and communicates the refinement with parameter servers, in parallel with other workers.

Ideally, to ensure high-quality refinement, the computation should use the up-to-date model parameters. This can be achieved through a Bulk Synchronous Parallel (BSP) implementation, where workers synchronize at the end of each iteration and will not proceed until the model parameters have been fully updated by *all* workers. However, BSP-style solution suffers from high synchronization overhead that makes it *impractical* to solve large ML problems [5], [10]: the presence of straggling workers inevitably slows down the entire learning progress.

Therefore, prevalent ML systems [5]–[9] employ an Asynchronous Parallel (ASP) model, where workers eagerly start the next iteration before receiving the latest parameters. In this way, the rate of update is maximized, leading to faster convergence than the BSP approach in many ML problems. However, ASP-style solution falls short of the quality of learned model. Without synchronization, the computation often iterate on stale parameters, which may drive the refinement away from the optimum [10]–[12].

Aiming at striking a balance between updates rate and updates quality in training, Stale Synchronous Parallel (SSP) model [6], [10], [13] is proposed recently as a middle ground between the ASP and BSP approach. In the SSP model, workers synchronize *only when* the *staleness* of parameters (measured by the number of missing updates from stragglers) exceeds a certain threshold. While this allows fast workers to use relatively fresher parameters, it provides little benefit for straggling machines which are left *unnoticed* about the refinements made by others. Consequently, updates generated by slowed machines may harm rather than benefit the training progress [12].

In this paper, we explore a new aspect to accelerate asynchronous distributed learning by enabling computation to iterate over fresher parameters for quality refinement. Our key observation is that, in ASP- and SSP-style solutions, a worker pulls fresh parameters from servers *only before the start of each iteration*. This restriction hides all the updates from other workers during the iteration, many of which are made shortly after the iteration begins. Should those updates be included, the quality of refinement would have been improved dramatically.

Following this observation, we propose a simple, yet effective parallel scheme, termed *speculative synchronization* (SpecSync), where each worker speculates about the parameter updates from others, and if necessary, it *aborts* the ongoing computation, pulls fresher parameters to *start over*, so as to opportunistically improve the quality of training. SpecSync offers two attractive properties. First, it effectively mitigates the inconsistency between the global parameters and their local replicas across workers—even slowed machines can timely discover the recent updates made by fast workers during computation. Second, it imposes no synchronization barrier and can be implemented on top of ASP and SSP models, well complementing existing approaches with improved performance.

However, there are two primary challenges need to be addressed. First, to facilitate SpecSync, workers need to be informed about the parameter updates from others. How to share this information without incurring high communication overhead is critical. Simply broadcasting each worker's update to others leads to all-to-all communications, which is too expensive to be useful in practice. Instead of broadcasting, we adopt a *centralized architecture* where workers report to a *centralized scheduler* once the parameter updates are pushed to servers. The scheduler oversees the global new pushes and notifies workers to re-synchronize for fresher parameters if necessary.

The second challenge arises as a question of when should a worker abort ongoing computation and perform re-synchronization? On one hand, to uncover more recent parameter updates, the worker should defer re-synchronization as much as possible. On the other hand, such deferral results in delayed computation, which in turn harms the training performance. We quantify the gain and loss due to speculation through a simple model. We formulate an optimization problem and propose an effective heuristic algorithm that strikes a good balance between uncovering more parameter updates and minimizing the computation delay.

We implemented SpecSync as a pluggable module in MXNet [7], a popular distributed ML systems with top-tier performance. Our implementation can be easily extended for other distributed ML systems and is open-sourced for public access.[1] We evaluated the effectiveness of SpecSync through cluster deployment in Amazon EC2 against three benchmarking ML applications: Matrix Factorization with MovieLens dataset [14], CIFAR-10 [15], and ImageNet [16]. The highlights of our evaluations are summarized as follows:

- SpecSync can significantly accelerate the training speed of different ML workloads (i.e., up to 3× faster) without compromising the training accuracy.
- SpecSync remains effective in a highly heterogeneous cluster and also scales with cluster size.
- SpecSync incurs little additional communication overhead during training.

II. Background and Motivation

In this section, we briefly introduce the background of distributed machine learning (ML) and the means to facilitate it in large clusters through the recently proposed Parameter Server (PS) architecture [5]–[9].

A. ML Problems Solved by Risk Minimization

In many learning problems, the input is a training dataset $\mathcal{D}$ consisting of n samples. A sample is a vector where each component characterizes a *learning feature*. Each sample x is associated with a label y. The objective of learning is to find a model with parameters w that correctly predicts label y given sample x. The learned model can then be used to predict y for any future x not seen in the training dataset.

[1] https://github.com/All-less/mxnet-speculative-synchronization

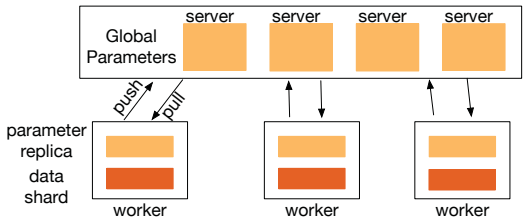


Fig. 1: The architecture of Parameter Server (PS).

To learn the model, the training algorithm solves a *risk minimization* problem. In particular, we define a loss function $l(x, y, w)$ that measures the prediction error (risk) if model w is used to predict label y given sample x. Our goal is to find the best model w that results in the minimum prediction errors over the entire training samples, i.e.,

$$\text{minimize}_w \sum_{x \in \mathcal{D}} l\,(x, y, w). \tag{1}$$

B. Parameter Server and Distributed SGD

To expedite training ML models on very large data sets, distributed ML systems have been proposed where the training is distributed over a cluster of commodity machines [5], [6], [8]. The recently proposed Parameter Server (PS) architecture can be employed to facilitate the system design [5]–[9].

As shown in Fig. 1, in the PS-based systems, training samples $\mathcal{D}$ are partitioned into a number of subsets $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_m$, each maintained by a *worker* machine. The model parameters are sharded across multiple *servers*, and can be accessed by all workers. Each worker also maintains a local replica of model parameters and iteratively refines it based on its own training samples. Periodically, workers push their local updates to servers, jointly refining the global parameters. The worker then pulls the most up-to-date parameters from servers and proceeds to the next iteration of training.

More precisely, workers perform *distributed SGD* (stochastic gradient descent) with *data parallelism* [17] in the PS-based ML systems. Each worker i iteratively passes over its own training samples $\mathcal{D}_i$. In each pass, the worker calculates the subgradient $\nabla l(x, y, w)$ for each sample x and updates the model with

$$w \leftarrow w - \eta \sum_{x \in \mathcal{D}_i} \nabla l(x, y, w), \tag{2}$$

where η is the *learning rate*. Every time a worker pushes an update, we say it finishes one *iteration*; when *all* workers finish an iteration, we say the training has gone through one *epoch*.

C. Synchronization Schemes

Asynchronous Parallel (ASP). Most popular PS-based ML systems [5]–[9] adopt Asynchronous Parallelism when perform distributed SGD, where each worker eagerly proceeds to the next iteration without waiting for the parameter updates from other workers. ASP-style execution fully exploits the computing cycles, maximizing the rate of update. However, the price paid is the *compromised quality* of learned models. In cluster environments, it is common to have workers process training samples at different speeds and make push and pull requests

at different rates. This results in *inconsistent* model replicas among workers. By the time a fast worker finishes one iteration and pulls parameters from servers, some slowed machines may remain in the middle of iteration. The fast worker hence misses the updates from those machines and proceeds to the next iteration with stale version of parameters. Training with stale parameters may poison the algorithm throughput as pointed in the literature [11], compromising training quality.

Bulk Synchronous Parallel (BSP) comes as an alternative scheme that enforces a consistent, up-to-date view of global parameters across workers. With the BSP scheme, workers synchronize at the end of each iteration and cannot proceed until all workers have pushed updates to servers. The BSP scheme ensures quality updates from each iteration, and is widely adopted in parallel analytics frameworks such as MapReduce [18], Spark [19], MLlib [20] and GraphX [21]. However, the BSP scheme incurs high synchronization overhead: fast workers must wait for stragglers to complete, wasting their computing cycles in idle. For this reason, the BSP scheme often performs poorly on large ML problems [10], [13].

Stale Synchronous Parallel (SSP) [6], [10], [13], [22] is proposed recently as a middle ground between the ASP and BSP schemes. In the SSP scheme, workers can asynchronously start next iteration with stale parameters, provided that the *staleness* (measured by the worker's progress ahead of the straggler) is within a bounded amount. The SSP scheme allows fast workers to timely discover updates from slowed machines, and is shown to provide a convergence guarantee as opposed to the ASP approach [10]. However, slowed workers may still lag behind with a rather inconsistent view to the global parameters. Prior work [12] shows that updates from slowed workers are often harmful rather than beneficial, especially when the parameters are coming close to the optimum. In fact, popular ML systems like TensorFlow [9] and MXNet [7] choose *not* to implement SSP based on the claim that only in rare cases can the improvement on training speed or convergence be observed [23].

To summarize, relaxed synchronization schemes can effectively speed up distributed learning, but at the cost of compromised quality due to inconsistent model replicas among workers [22]. This motivates us to propose a new approach to accelerate asynchronous distributed learning by enabling more up-to-date views of global parameters across workers.

III. Staying Fresh through Naïve Waiting

In this section, we examine the behaviors of asynchronous learning through empirical studies and explore a new aspect to keep parameters fresh in computation.

A. Pushes after a Pull: The Source of Staleness

Without synchronization, each worker eagerly pulls parameters to start next iteration, and will miss the following pushes made by others before the next pull. That is, asynchrony hides *pushes after a pull* (PAP)—the main source where staleness derives. Fig. 2 illustrates an example. The parameters worker-1 pulls at time t_1 have received only one update from itself, while the ones worker-2 pulls include four updates from all workers and are much fresher. In fact, worker-1 has the most outdated parameters, as it misses more PAP than anyone else.

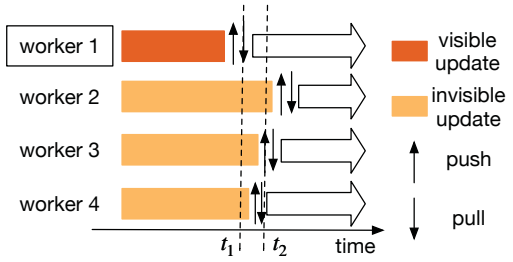


Fig. 2: Asynchrony hides pushes after a pull (PAP). Worker-1 misses more PAP than anyone else and ends up with the most outdated parameters.

Intuitively, to stay fresh, a worker should uncover more recent pushes made by others. A simple way to do so is to *defer the pull request by a small amount of time*, so as to capture more recent pushes that are otherwise invisible. In the previous example, delaying the pull request of worker-1 to t_2 exposes the two pushes from worker-3 and worker-4, enabling a more up-to-date view of the global parameters.

However, the deferral of a pull request inevitably delays the start of an iteration, leading to longer completion time. Without a careful control, the harm caused by the delay may outweigh the benefits derived from fresher parameters. Therefore, whether deferring a pull request is beneficial critically depends on how many pushes are expected to be made shortly after the pull.

We investigate this issue through an empirical study using a real ML application. We trained a deep residual network with CIFAR-10 dataset [15] in MXNet [7], deployed in an Amazon EC2 cluster consisting of 40 `m4.xlarge` instances.[2] The training is performed in an ASP model. We collected the workload traces and analyze how many pushes were made by others between two consecutive pulls of a worker (i.e., the number of missing updates in one iteration). In our experiment, an iteration typically spans around 14 seconds. We further divide an iteration into a number of 1-second intervals (i.e., 0-1s, 1-2s, etc.), and for each interval, we count the number of PAP received in it. We show the distribution of each interval as a box plot in Fig. 3, where boxes depict 25th, 50th, and 75th percentiles, and whiskers depict 5th and 95th percentiles. We observe approximately uniform arrivals of PAP in each interval. In particular, if we focus on the number of pushes received within two seconds after a pull is made (the first two boxes in Fig. 3), the median is over 6. That is to say, by delaying the pull request by 14% of the iteration time, we can allow 50% of workers to include at least 15% of recent updates. These promising numbers suggest that a slight delay of iteration is sufficient to discover more fresh parameters.

[2]In MXNet, each node is both a worker and a server.

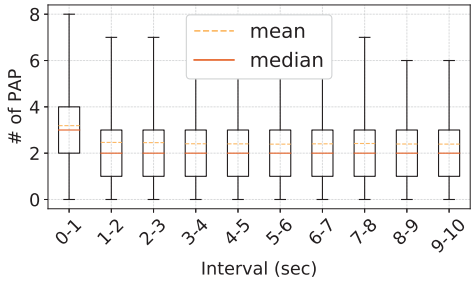


Fig. 3: Distribution of the number of pushes received in a time interval after a pull is made (PAP). Each interval spans one second.

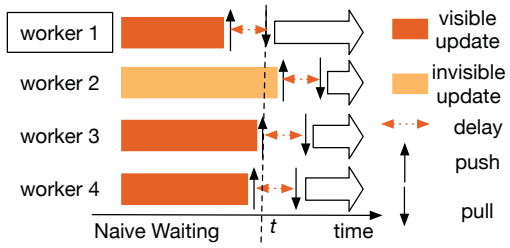


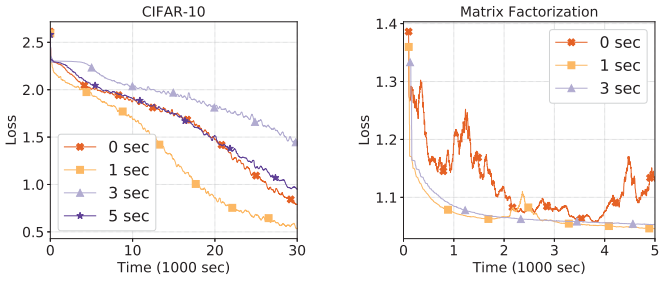
Fig. 4: Naïve waiting defers each pull request by a short period of time, allowing worker-1 (and others) to uncover more updates than it could have had in Fig. 2.

B. Naïve Waiting

Motivated by our empirical studies, we propose a simple strategy which we call *naïve waiting*. As the name suggests, each worker simply delays its pull requests by a short period of time, so as to uncover more pushes that are otherwise invisible. Fig. 4 illustrates naïve waiting applied to the example in Fig. 2. We see that a slight delay of pulls allows *all* workers to include more updates in their parameters than they could in Fig. 2: worker-1 now uncovers three updates (highlighted in Fig. 4), while the other three workers get all four (not shown in Fig. 4).

To quantify the benefits of naïve waiting in real-world systems, we implemented it in MXNet 0.7 [24] and evaluated its performance in an EC2 cluster composed of 40 `m4.xlarge` instances. We ran two benchmarking ML workloads: a deep residual network with CIFAR-10 dataset [15] and matrix factorization with MovieLens dataset [14]. For each workload, we configured naïve waiting with different delays and compared their impacts to the performance. Fig. 5 depicts the learning curves of the two ML workloads. By delaying each pull request by 1 second, both workloads achieve significant performance improvements. However, as the delay increases, more computing cycles get wasted, and the performance deteriorates. As illustrated by the CIFAR-10 workload in Fig. 5a, delaying each pull request by 3 seconds yields little benefit over the original implementation; further increasing the delay to 5 seconds even does more harm than good.

To summarize, our experiments confirm that simply deferring each pull request to expose more fresh parameters can be beneficial. However, such a benefit is conditioned on finding the "right" delay time, which by itself is technically non-trivial. In fact, naively deferring each pull request may not always be justified—more often than not, the delay may expose only a few new updates. We therefore give up on naïve waiting and turn to a new approach to avoid unjustified delay.



(a) CIFAR-10 (b) Matrix Factorization

Fig. 5: Convergence curves of naïve waiting with different delay times. Curves with zero delay correspond to the stock MXNet implementation.

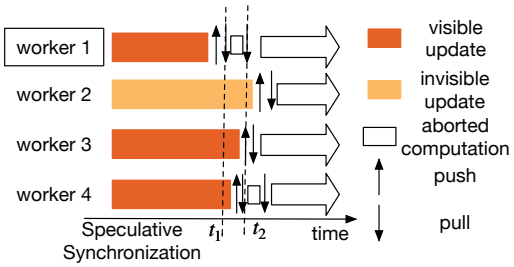


Fig. 6: Illustration of speculative synchronization. Worker-1 speculatively aborts computation after observing the two pushes made by two peers. It pulls parameters again and starts over.

IV. Speculative Synchronization

In this section, we present a new scheme, called *speculative synchronization* (SpecSync), which allows workers to speculatively abort the ongoing computation and start over with fresher parameters. We model the gain and loss due to speculative re-execution, based on which we propose an effective heuristic algorithm to determine when to restart computation for workers. SpecSync can be implemented in both ASP and SSP models, *complementing* existing solutions with improved performance.

A. Overview

Key idea. Instead of imposing an arbitrary delay without justification, we let the worker asynchronously proceed to the next iteration immediately, while at the same time speculating about the updates made by others. Once the worker learns that the global parameters have been updated "enough" times, it will abort the ongoing iteration, pull the fresher parameters to start over—if that is not too late yet.

Continuing the example in Fig. 2, we apply speculative synchronization and illustrate workers' behaviors in Fig. 6. We start to focus on worker-1. After finishing the first iteration,

it pulls parameters and starts the next iteration immediately. Shortly after it starts, it learns that two other peers have pushed updates to servers (highlighted in Fig. 6), which it views as a significant-enough change made to the global parameters. Worker-1 hence aborts the ongoing iteration, re-synchronizes with servers to include those two recent updates, and starts over with much fresher parameters. The abort-and-restart decision is also made by worker-4 upon its notice of two updates pushed shortly after the second iteration starts. In contrast, workers 2 and 3 choose not to restart as they do not see enough updates (two in this example) pushed to servers since their last pulls.

Benefits. SpecSync offers two benefits.

1) It avoids unjustified delays, minimizing the cost of wasted computing cycles due to abortion. With SpecSync, an iteration gets restarted only when the global parameters have undergone *significant enough* updates within a short period of time after the iteration begins. When that happens, the improved quality of refinement brought by fresher parameters will surely outweigh the cost of slightly delayed computations.

2) SpecSync can be flexibly implemented in both ASP and SSP models, complementing *them with improved performance.* In fact, the only difference between ASP and SSP is that the latter enforces bounded staleness in that fast workers must wait for slowed ones to catch up. With SpecSync implemented in the SSP model, workers can *actively* seek opportunities to restart computation with fresher parameters, before the staleness bound is reached. This also gives slowed workers a chance to timely capture updates from fast peers by aborting computations, ensuring a more consistent view of global parameters. As a result, the quality of updates generated by straggling workers can be improved dramatically.

Challenges. However, to facilitate SpecSync, there are two major challenges we should address.

1) How can we efficiently notify each worker when the global parameters are updated by others, without incurring high communication overhead? Simply broadcasting each worker's push notification to others causes all-to-all communications, and is too expensive to implement. We shall address this challenge in Sec. V through a *centralized* system architecture, where a *scheduler* oversees pushes from all workers and notifies each when the global parameters have been updated sufficient times since its last pull.

2) As the global parameters are refined, how can we determine, for each worker, when to abort computation and start over? We employ a simple speculation strategy with two hyperparameters: `ABORT_TIME` and `ABORT_RATE`. In particular, after starting an iteration, a worker speculates about the parameter updates made in a time period of length `ABORT_TIME`. The worker counts the number of pushes made by others in that speculation period, and normalizes the count by the total number of workers to obtain the *push rate*. If the push rate exceeds `ABORT_RATE`, the worker is convinced that the global parameters have undergone significant updates since its last pull. The worker then aborts the ongoing computation, re-synchronizes with servers, and starts over.

The choices of the two hyperparameters critically determine the performance of speculative synchronization. The question is: given a learning workload, how can we judiciously choose the two hyperparameters that lead to the optimal performance? We answer this challenge next.

B. Adaptive Hyperparameter Tuning

To achieve the optimal performance of SpecSync, we propose a heuristic algorithm that adaptively tunes the two hyperparameters. We start with a problem formulation.

Problem formulation. Suppose that worker i restarts an iteration after a speculation period. Worker i gains by uncovering more recent updates that are otherwise invisible, at the expense of a delayed computation. Such a delay may hide worker i's update from being captured by others. By the time worker i pushes an update, it would be too late for some other workers to capture that refinement, which in turn harms the parameter freshness of those workers.

In our model, we measure the *freshness gain* by the number of updates others pushed in the speculation period, and the *freshness loss* by the number of *missed peers*, which are workers that missed the (delayed) push of the speculator. More precisely, we divide time into *epochs* (cf. Sec. II-B). Let Δ denote the `ABORT_TIME` used in epoch τ. For worker i, let $u_{i,\tau}(\Delta)$ be the number of updates it uncovers during its speculation period. The value of $u_{i,\tau}(\Delta)$ measures the freshness gain. Let $l_{i,\tau}(\Delta)$ be the number of missed peers, which quantifies the freshness loss. Worker i hence makes a *freshness contribution* $u_{i,\tau}(\Delta) - l_{i,\tau}(\Delta)$. To make a positive contribution, worker i aborts the computation only when the gains outweighs the loss, i.e., $u_{i,\tau}(\Delta) \geq l_{i,\tau}(\Delta)$. We shall discuss later how this can be achieved by correctly choosing an `ABORT_RATE`.

Summing up the contribution over all workers, we obtain the *overall freshness improvement* due to speculative synchronization, i.e.,

$$F_\tau(\Delta) = \sum_{i=1}^{m} \left(u_{i,\tau}(\Delta) - l_{i,\tau}(\Delta)\right). \tag{3}$$

Our goal is to find the optimal Δ at the beginning of each epoch that maximizes the overall freshness improvement:

$$\text{maximize}_\Delta F_\tau(\Delta). \tag{4}$$

Estimating gain and loss. Unfortunately, Problem (4) cannot be directly solved in practice, as the exact computation of $u_{i,\tau}(\cdot)$ and $l_{i,\tau}(\cdot)$ requires knowing the complete push/pull sequence in epoch τ before the epoch starts. We therefore turn to estimations of the gain and loss.

In particular, to estimate how many updates worker i will uncover after a speculation period Δ, we simply refer back to the previous epoch and count the number of updates the worker would have uncovered if it had deferred its last iteration by Δ. More precisely, we estimate $u_{i,\tau}(\Delta)$ as

$$\tilde{u}_{i,\tau}(\Delta) = u_{i,\tau-1}(\Delta), \tag{5}$$

where $\tilde{u}_{i,\tau}(\cdot)$ is an estimation of $u_{i,\tau}(\cdot)$. Our insight is that the algorithmic behaviors and machine performance are usually

stable in a short period of time. Therefore, the freshness gain computed based on the push history in the previous epoch can be used as a good approximation to that in the current epoch.

Ideally, we can use the same approach to estimate freshness loss, i.e., $\tilde{l}_{i,\tau}(\Delta) = l_{i,\tau-1}(\Delta)$. However, simply using the push history in epoch $\tau - 1$ may not be sufficient to compute $l_{i,\tau-1}(\Delta)$. In fact, assuming the worker deferred its last iteration by Δ, it is possible that the iteration would still be running now, and the freshness loss caused by that delay cannot be fully characterized until the iteration completes.

Unable to make use of historical traces, we simply estimate freshness loss $l_{i,\tau}(\Delta)$ as *the expected number of missed peers assuming uniform arrivals of pull requests*. While this technical assumption may not always hold in practice, it simplifies the analysis and makes the problem tractable. We shall show in Sec. VI that despite this technical assumption, our solution can still achieve near-optimal performance in a number of ML applications.

Specifically, let $T_{i,\tau}$ be the iteration span of worker i in epoch τ, which can be accurately predicted from history. Deferring the iteration by Δ hides the worker's update from being captured by another with probability $\frac{\Delta}{T_{i,\tau}}$. Therefore, the expected number of missed peers, or the estimate of freshness loss, is

$$\tilde{l}_{i,\tau}(\Delta) = \frac{\Delta}{T_{i,\tau}}(m-1). \tag{6}$$

Substituting the gain and loss by their estimates, we obtain an estimate of the overall freshness improvement:

$$\tilde{F}_{\tau}(\Delta) = \sum_{i=1}^{m} \left(\tilde{u}_{i,\tau}(\Delta) - \frac{m-1}{T_{i,\tau}}\Delta \right). \tag{7}$$

Hyperparameter tuning. Directly searching the optimal Δ to maximize improvement estimate $\tilde{F}_{\tau}(\Delta)$ can be difficult, as Eq. (7) is non-convex. We present an efficient algorithm that optimally solves this problem. Our key observation is that, to maximize Eq. (7), it is sufficient to search *a finite number* of Δ. To see this, consider worker i, and let Δ gradually increase from 0. By definition, $\tilde{u}_{i,\tau}(\Delta)$ increments only when the increase of speculation period exposes one more push from another worker. Therefore, $\tilde{u}_{i,\tau}(\Delta)$ is a *step function*. Also note that the estimate of freshness loss (cf. Eq. (6)) linearly increases with Δ. Putting them together, the freshness contribution (gain minus loss) reaches the maximum *only if* the speculation interval *right aligns* with a push. It is easy to see that there are $O(m^2)$ such intervals in total, where m is the number of workers. We can therefore enumerate all candidate Δ and choose the one that maximizes Eq. (7). Algorithm 1 illustrates the details, with time complexity $O(m^3)$.

Once the optimal Δ^* is found (`ABORT_TIME`), we set `ABORT_RATE` so that workers abort computation only when the gain outweighs the loss. Specifically, let Γ denote the `ABORT_RATE` used by worker i. By our scheme, worker i aborts computation if the number of updates received during the speculation period exceeds Γm. We therefore set $\Gamma = \tilde{l}_{i,\tau}(\Delta^*)/m$ to ensure that re-execution always makes a positive contribution to the overall freshness improvement.

Algorithm 1 Adaptive Tuning

```
– m: number of workers
– T: averaged iteration span
function TUNEPARAM
    l ← sequence of all pushes made in the last epoch
    {Δ} ← time difference between all pairs of pushes in l
    for Δ ∈ {Δ} do
        F ← computation result of Eq. (7)
        if F is maximal then
            ABORT_TIME ← Δ
            ABORT_RATE ← Δ(m − 1)/Tm
```

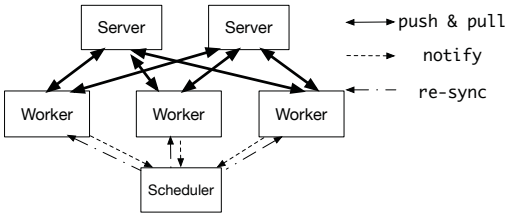


Fig. 7: The architecture of speculative synchronization.

The experimental evaluation in Sec. VI verifies that our heuristic approach achieves *near-optimal* speedup for many ML applications even compared with the scheme using the optimal hyperparameters cherry-picked via exhaustive search.

V. Implementation

In this section, we present our implementation of SpecSync atop MXNet [7], [24], a popular ML framework employing the parameter server (PS) architecture. While we base our implementation on MXNet, nothing precludes it from being ported to other PS-based systems such as TensorFlow [9].

A. Architecture Overview

Centralized implementation. We employ a *centralized implementation* for SpecSync. As shown in Fig. 7, our implementation consists of three components: workers, servers, and a centralized scheduler. At the beginning of an iteration, a worker pulls model parameters from servers and starts computation. In the mean time, the worker *delegates* the speculation job to the centralized scheduler which oversees the parameter updates from others and notifies the worker when it is time to re-synchronize. Upon receiving an instruction from the scheduler, the worker pulls fresh parameters from servers and restarts the computation. After an iteration completes, the worker pushes the computed update to servers, and notifies the scheduler. This way, the scheduler can keep track of pushes made by all workers, enabling it a global view to perform speculation for each worker.

Benefits. The centralized implementation offers two benefits over a direct implementation where each worker performs speculation *individually*. First, it eliminates the need for all-to-all communication among workers, which is unavoidable in the direct implementation as each worker must *broadcast* a push notification to all others. In contrast, workers in the centralized architecture only report to the scheduler. We shall show in Sec. VI that the communication overhead incurred

Algorithm 2 Speculative Synchronization

Workers: $i = 1, 2, \ldots, m$

1: **function** WORKERSTARTSEPOCH(e)
2: Pull model parameters $w^{(e,0)}$ from servers
3: **for all** training batch $t = 0, 1, 2, \cdots, T$ **do**
4: Start computing gradient $g_i^{(e,t)}$ in a non-blocking manner
5: **if** worker receives `re-sync` during computation **then**
6: Abort computation and pull $w^{(e,t)}$ from servers
7: **go to** 4 ▷ Restart compution
8: Push $g_i^{(e,t)}$ to servers
9: Pull $w^{(e,t+1)}$ from servers
10: Send `notify` to scheduler

Scheduler:

– l: a list of timestamps of all pushes
1: **function** STARTSTRAINING
2: **for** epoch $e = 0, 1, 2, ..., E$ **do**
3: Issue WORKERSTARTSEPOCH(e) to all workers
4: **function** HANDLENOTIFICATION(`msg`)
5: Append current timestamp to l
6: Call CHECKRESYNC(`msg.sender`) after `ABORT_TIME`
7: **function** CHECKRESYNC(`senderId`)
8: `cnt` $\leftarrow$ number of pushes received within `ABORT_TIME`
9: **if** `cnt` $\geq m\times$ `ABORT_RATE` **then** ▷ Time to re-synchronize
10: Issue `re-sync` to the worker with `senderId`

among workers and the scheduler is negligible. Second, in the centralized architecture, only the scheduler needs to maintain a global view of the push history of workers. However, in the direct implementation, this information must be separately maintained by each worker, leading to unnecessary storage redundancy.

B. Workflow

We elaborate on the implementation of each component, following the workflow of SpecSync illustrated in Algorithm 2.

Workers communicate with the scheduler through two dedicated messages: `notify` and `re-sync`. A `notify` message simply contains a `senderId` and is sent to the scheduler by a worker upon the completion of an iteration. The `notify` message triggers the speculation on the scheduler. At the same time, the worker pulls parameters from servers and proceeds to the next iteration, during which the worker expects a `re-sync` message from the scheduler to re-synchronize with servers and start over. Once the iteration completes, the worker pushes update to servers, sends a `notify` message to the scheduler, and repeats the entire process.

Servers are *agnostic* to speculative synchronization performed by workers and the scheduler. Their behaviors remain the same as in the stock MXNet, and is not shown in Algorithm 2.

Scheduler keeps track of pushes made by workers through `notify` messages and performs two jobs. First, it computes the two hyperparameters `ABORT_TIME` and `ABORT_RATE` at the beginning of each epoch using Algorithm 1. Second, it implements the logic of SpecSync *on behalf of each worker*. Specifically, it maintains a *push counter* and a *timer* for each worker. Upon receiving a `notify` message from a worker, the scheduler appends it to the push history and kicks start the speculation for the sender. To do so, it resets the push counter of the sender and starts the corresponding timer which will expire in `ABORT_TIME`. During the speculation period, the push counter increments whenever a `notify` message is received. Upon the timer expires, the scheduler checks whether the counter is more than $m\times$`ABORT_RATE`, where m is the number of workers. If so, the scheduler instructs the worker to re-synchronize through a `re-sync` message, as enough updates have been pushed since the worker's last pull.

TABLE I: Models and datasets used for workloads. The iteration time is based on `m4.xlarge` instance.

workload	# parameters	dataset	dataset size	iteration time
MF	4.2 million	Movielens	100,000	3s
CIFAR-10	2.5 million	CIFAR-10	50,000	14s
ImageNet	5.9 million	ImageNet	281,167	70s

VI. EVALUATION

We conduct extensive experimental evaluations to validate the effectiveness and robustness of the proposed SpecSync. We first compare the accuracy and runtime of SpecSync with the default synchronization scheme of MXNet. Then we demonstrate the robustness of SpecSync in terms of handling heterogeneity and scalability. We examine the communication overhead afterward. Finally, we discuss the advantage of hyperparameter tuning with respect to the algorithm efficiency and adaptivity.

A. Experiment Setup

Workloads. We use three different workloads to drive the experiments for evaluation, and we summarize them in Table I. For the first workload, we use MovieLens [14] as the dataset to train a recommendation system with matrix factorization (MF), the batch size is set to 100,000. We train a 110-layer deep residual network [25] with CIFAR-10 dataset as the second workload. To achieve the best performance, we set the batch size to 128 and let the learning rate decrease from an initial value 0.05 at epochs 200 and 250, respectively [26]. Finally, we train an 18-layer deep residual network with ImageNet dataset [16]. The batch size is set to 128 and the learning rate is set to 0.3. We choose the above workloads because they represent different characteristics of popular ML applications. For example, the input data of CIFAR-10 and ImageNet are pictures represented by dense vectors, while the input data of MF are user ratings represented by sparse vectors. The training data set size and model size of the above workloads are also representative from small to large applications with iteration time spanning from a few seconds to more than one minute.

Testbed. We build clusters on Amazon EC2 for running experiments. We use a 40-node cluster (**Cluster 1**) for effectiveness evaluation, which is composed of 40 `m4.xlarge` instances. This cluster is for the scenario of asynchronous distributed learning on homogeneous hardware. We build a heterogeneous cluster (**Cluster 2**) for evaluating SpecSync's ability to deal with heterogeneity, which consists of 10 `m3.xlarge`, 10 `m3.2xlarge`, 10 `m4.xlarge`, and 10 `m4.2xlarge` instances.

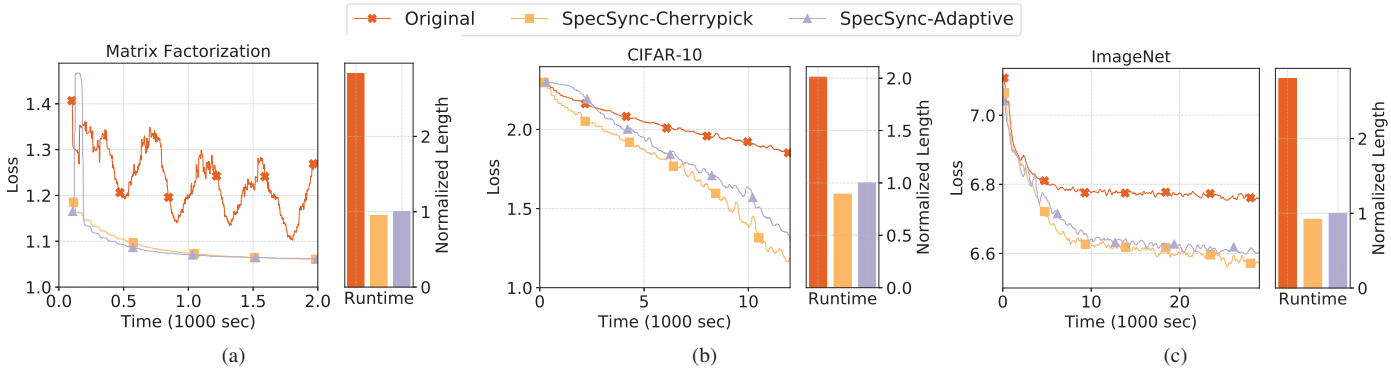


Fig. 8: Loss (left plot in each sub-figure) and runtime (right plot in each sub-figure) comparison of (a) MF, (b) CIFAR-10, and (c) ImageNet. For better visualization, we only show the results up to when one of the approaches is converged (i.e., loss is below the target value for 5 consecutive iterations).

To evaluate SpecSync's scalability, we use 3 clusters with 20, 30 and 40 `m4.xlarge` instances respectively. All the instances run Ubuntu Server 16.04 LTS, and are configured to run our extended MXNet 0.7.

Schemes. We use three different synchronization schemes in our evaluation. (1) **Original**: default asynchronous synchronization scheme provided by MXNet. (2) **SpecSync-Cherrypick**: the cheerypick version (i.e., tune the `ABORT_TIME` and `ABORT_RATE` hyperparameters using grid search) of the proposed speculative synchronization, which is built on top of the asynchronous synchronization scheme provided by MXNet. (3) **SpecSync-Adaptive**: the adaptive version (i.e., tune the hyperparameters adaptively) of the proposed speculative synchronization, which is also implemented based on the asynchronous synchronization scheme provided by MXNet.

B. Effectiveness of SpecSync

We first evaluate the effectiveness of the proposed SpecSync on Cluster 1 with three different workloads outlined in Table I. We report two metrics: loss change over time (for accuracy evaluation) and runtime to convergence (for runtime performance evaluation), and present the results in Figure 8. Runtime is measured as the timespan from the beginning of training to convergence, where convergence is defined as the loss staying below the target value for 5 consecutive iterations. By comparing the results of different schemes, it is clear that the proposed SpecSync can significantly speed up the training process, i.e., up to 2.97× speedup for MF, up to 2.25× speedup for CIFAR-10, and up to 3× speedup for ImageNet. The results also demonstrate that even though SpecSync-Adaptive is not able to achieve the same performance as SpecSync-Cherrypick, the difference is very small, which verified the effectiveness of the proposed adaptive algorithm. We also notice for SpecSync-Adaptive, the learning curves (loss as a function of time) has more jitter at the beginning than SpecSync-Cherrypick, this is due to the adaptive nature of SpecSync-Adaptive.

With SpecSync, when re-synchronization occurs during an iteration, the length of the iteration becomes longer, but the training quality of the iteration is improved due to the fresher parameters used for training, so the overall training speed becomes faster. To demonstrate this, we plot the loss as a function of the iteration number and also the accumulated iteration number for different schemes in Fig. 9. The comparison results shows that it takes up to 58% fewer iterations for SpecSync to converge compared with Original. Such an improvement suggests that with SpecSync, the quality of computing is improved while the efficiency of utilizing the distributed computing power by asynchronous learning is preserved.

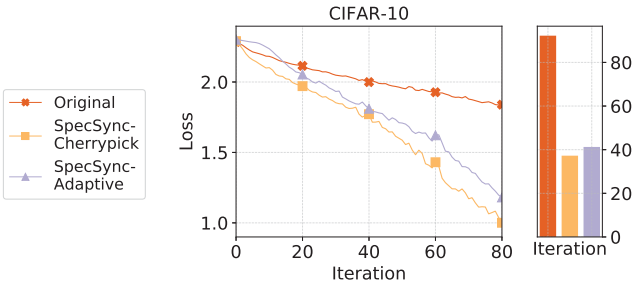


Fig. 9: Loss as a function of iteration number (left plot) and accumulated iteration number (right plot) for different synchronization schemes using CIFAR-10.

C. Robustness of SpecSync

Heterogeneity. In distributed machine learning, heterogeneity (caused by various factors from hardware resources to software failures) can result in inconsistent training progress among workers and therefore slows down the training speed [12]. Here we conduct experiments using a heterogeneous cluster (Cluster 2), which consists of 4 different instance types, to evaluate SpecSync's robustness in the presence of heterogeneity. In the interest of space, we only show the results of CIFAR-10 in Fig. 10. From the loss plot, it is clear that SpecSync-Adaptive outperforms Original in both homogeneous and heterogeneous clusters.[3] The results also verified that the heterogeneity can

[3]Given we have already verified the difference between SpecSync-Adaptive and SpecSync-Cherrypick is small, for clear presentation, we do not show the results of SpecSync-Cherrypick in all following plots.

slow down the training speed. Heterogeneity slows down BSP because it increases synchronization overhead as fast workers need to wait for stragglers to complete in each iteration. ASP and SSP also suffer from heterogeneity because the staleness gap between workers is intensified due to the mismatch in training speed, which makes the convergence slower. SpecSync-Adaptive actively improves the freshness of parameter replica to alleviate inconsistency between workers compared to ASP and SSP, and does not have the synchronization issue of BSP, thus being more robust to heterogeneity. Another interesting observation is the speedup of SpecSync-Adaptive over Original is smaller compared to the same experiments on homogeneous cluster as shown in Fig. 8b. This is because our adaptive tuning approach assumes uniform arrivals of pull requests, but in heterogeneous environment, the arrival becomes less uniform, so the quality of the tuned hyperparameters deteriorates.

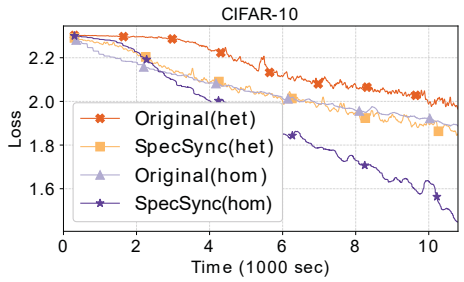


Fig. 10: Loss comparison for different synchronization schemes running CIFAR-10 in both homogeneous and heterogeneous clusters.

Scalability. We perform sensitivity analysis for cluster size to evaluate the scalability of the proposed SpecSync-Adaptive. We show results for two scenarios that are commonly used in practice. The first scenario is when machine learning practitioners with clear training target in mind. So we demonstrate the speedup of SpecSync-Adaptive over Original in runtime for achieving the same target training accuracy of CIFAR-10 using cluster size of 20, 30, and 40, respectively (the left plot of Fig. 11). The second scenario is usually for the case with fixed budget and aims to achieve the best training accuracy under the given budget (e.g., rent cloud instances for a time period that can be fit into the given budget). Therefore, we compare the loss improvement of SpecSync-Adaptive over Original when training CIFAR-10 for the same amount of time using different cluster sizes as shown in the right plot of Fig. 11. It is clear that in both scenarios, SpecSync-Adaptive consistently outperforms Original running with different cluster size. More importantly, when the cluster size grows, the improvement becomes even larger. This suggests that SpecSync-Adaptive has better scalability than Original.

D. Communication Overhead

SpecSync may introduce extra communication overhead due to exchanging additional information between workers and parameter servers. Fig. 12 reports the accumulated data transfer

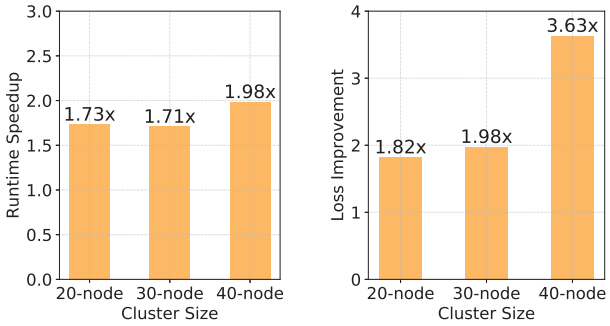


Fig. 11: Runtime speedup for achieving the same training accuracy target (left plot) and loss improvement with the same training time (right plot) for SpecSync-Adaptive over Original using CIFAR-10 as workload under different cluster size.

over time for different workloads using Original and SpecSync-Adaptive. It is clear that the accumulated data transfer is very close between SpecSync-Adaptive and Original at all time, meaning there is very little additional bandwidth consumed by SpecSync-Adaptive. In addition, the overall training time is shorter using SpecSync-Adaptive, so the total data transfer can be actually smaller compared to Original, i.e., compare the right most points in Fig. 12. Take CIFAR-10 as an example, Original incurs a total data transfer of 3.17 TB while SpecSync-Adaptive only needs to transfer 2.00 TB in total, a saving of nearly 40% of communication overhead.

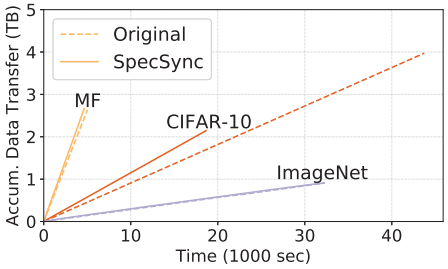


Fig. 12: Accumulated data transfer over time for different workloads under different schemes.

We also refer to Fig. 13 for the breakdown of overall data transfer incurred by SpecSync-Adaptive in three parts: push and pull, which are the regular communication traffic (i.e., also in Original); the additional pulls triggered by re-synchronization; notifications including `notify` and `re-sync` messages. The results indicate that the additional overhead introduced by notification messages is marginal given the message size is quite small. The main additional communication overhead is introduced by re-synchronization. However, such overhead is compensated by the improved quality of training, thanks to the fresher parameters. In general, for computation-bound workloads, SpecSync can achieve higher computation efficiency as the training quality improves with fresher parameters, and therefore alleviates the computation bottleneck impact. For communication-bound workloads, SpecSync automatically adjusts the speculative synchronization hyperparameters (e.g.,

ABORT_TIME and ABORT_RATE) so that the re-synchronization is performed more conservatively to balance the freshness and network performance.

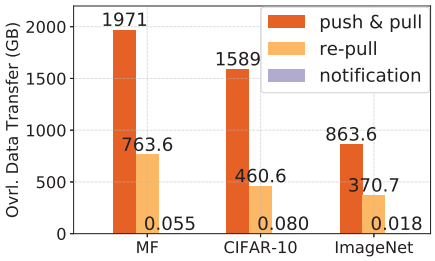


Fig. 13: Overall data transfer breakdown for SpecSync-Adaptive.

E. SpecSync-Cherrypick vs. SpecSync-Adaptive

We evaluate SpecSync-Adaptive here against SpecSync-Cherrypick in terms of the overhead of tuning hyperparameters. For SpecSync-Adaptive, there is little overhead as it simply searches best hyperparameter through logged information (a short list of numbers) using a closed-form estimation function (Eq. (7)), no additional profiling experiment is needed. For SpecSync-Cherrypick, the hyperparameters are tuned through exhaustive search with profiling experiments. Examples of total search time for different workloads are shown in Table II. In the example, we search 10 different values of ABORT_RATE for each workload. For ABORT_TIME, we try to minimize the number of trials by restricting the search range and increasing search step (e.g., we use half of the batch time as upper bound and make sure the search step is greater than communication time). It is clear that even with reasonable search range and step, the cost is still very high as each experiment takes long time, and the accumulated cost becomes even higher. Therefore, SpecSync-Adaptive has much lower search overhead for hyperparameter tuning. In addition, SpecSync-Adaptive adapts the hyperparameters for each iteration, which is more robust than the fixed value solution of SpecSync-Cherrypick.

TABLE II: Cost of hyperparameter exhaustive search using SpecSync-Cherrypick.

workload	# of trial for ABORT_TIME	# of trial for ABORT_RATE	each trial time (hour)	total search time (hour)
MF	3	10	1.33	40
CIFAR-10	7	10	6	420
ImageNet	10	10	> 8	> 800

F. Discussion

The experimental study in this section is mainly based on CPU cluster. For GPU or other architecture based cluster, SpecSync is also compatible because the synchronization happens at node level, where the node can be a virtual concept, e.g., it can be a machine with CPU or GPU, and/or it can be a CPU or GPU within a multi-CPU/GPU machine. In the interest of space, we leave the detailed study for clusters with different hardware architecture as our future work. We also leave experimental study of other machine learning applications and other machine learning frameworks as our future work.

VII. Related Work

Many recent works have been proposed to improve the performance of existing synchronization schemes employed in distributed ML systems. Notably, for the SSP-style execution, Wei et al. [27] proposed a system called Bösen to maximize the communication efficiency by prioritizing update messages that are the most significant to the final convergence. Harlap et al. [28] proposed FlexRR to mitigate the straggler problem by dynamically offloading the training work of straggling workers to fast machines. For the ASP-style executions, Zhang et al. [29] proposed a staleness-aware asynchronous SGD algorithm that achieves guaranteed convergence by dynamically adjusting the learning rate based on the gradient staleness. Jiang et al. [12] applied the similar approach to heterogeneous clusters, where slowed workers are assigned lower learning rate to alleviate the negative impact of their updates. More recently, Chen et al. [30] revisited the BSP model and suggested synchronous training of deep models with *backup* workers, so as to mitigate the impact of stragglers. All those techniques are orthogonal to our proposal and can be combined together with SpecSync.

SpecSync is inspired by *delay scheduling* [31] used in MapReduce clusters, where map tasks prefer to running on machines that can provide data locality. With delay scheduling, the assignment of a task is delayed for a short period of time if its data locality cannot be satisfied at the current moment. Unlike delay scheduling in MapReduce where the delay is almost surely beneficial [31], we have shown in Sec. III that naïvely delaying each pull request is not always justified (Fig. 5). SpecSync is proposed to address this problem. In fact, aborting and restarting a task is considered expensive for MapReduce jobs, and is usually not an option to the scheduler.

VIII. Concluding Remark

In this paper, we have investigated a new aspect to improve parameter freshness for asynchronous distributed learning. We have proposed SpecSync where each worker speculates about the parameter updates pushed by others after an iteration starts. In case that a sufficient number of pushes have been made within a short period of time, the worker aborts the ongoing iteration, re-synchronizes with servers, and starts over with fresher parameters. We have designed an adaptive hyperparameter tuning algorithm to judiciously determine the span of speculation period and the threshold number of pushes beyond which a re-synchronization is triggered. We have implemented SpecSync atop MXNet in a centralized architecture with little extra communication overhead. Experimental evaluations through EC2 deployment demonstrate that SpecSync achieves up to 3× speedup in three benchmarking ML workloads.

Acknowledgment

This work was supported in part by ITF Award ITS/391/15FX and NSF Grant CCF-1756013. Chengliang and Huangshi were supported in part by the Hong Kong PhD Fellowship Scheme.

REFERENCES

[1] D. Talbot, "How Microsoft Cortana improves upon Siri and Google Now," http://www.tomshardware.com/news/microsoft-cortana-unique-features, 26506.html, accessed: 2015-11-20.

[2] C. Rosenberg, "Improving photo search: A step across the semantic gap," http://googleresearch.blogspot.com/2013/06/improving-photo-search-step-across.html, accessed: 2015-11-20.

[3] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan, "A picture is worth a thousand (coherent) words: building a natural description of images," http://googleresearch.blogspot.com/2014/11/a-picture-is-worth-thousand-coherent.html, accessed: 2015-11-20.

[4] F. Nelson, "Nvidia demos a car computer trained with âĂIJdeep learning"," http://www.technologyreview.com/news/533936/, accessed: 2015-11-20.

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

[6] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter serve." in *USENIX OSDI*, 2014.

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

[8] T. M. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project Adam: Building an efficient and scalable deep learning training system," in *USENIX OSDI*, 2014.

[9] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "TensorFlow: A system for large-scale machine learning," in *USENIX OSDI*, 2016.

[10] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More effective distributed ML via a stale synchronous parallel parameter server," in *NIPS*, 2013.

[11] J. Langford, A. J. Smola, and M. Zinkevich, "Slow learners are fast," in *NIPS*, 2009.

[12] J. Jiang, B. Cui, C. Zhang, and L. Yu, "Heterogeneity-aware distributed parameter servers," in *ACM SIGMOD*, 2017.

[13] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons *et al.*, "Exploiting bounded staleness to speed up big data analytics." in *USENIX ATC*, 2014.

[14] F. M. Harper and J. A. Konstan, "The movielens datasets: History and context," *ACM Trans. Interactive Intelligent Sys.*, vol. 5, no. 4, p. 19, 2016.

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

[16] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *IEEE CVPR*, 2009, pp. 248–255.

[17] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized stochastic gradient descent," in *NIPS*, 2010.

[18] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *USENIX OSDI*, 2004.

[19] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *USENIX NSDI*, 2012.

[20] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, "MLlib: Machine learning in apache spark," *J. Mach. Learn. Res.*, vol. 17, no. 1, pp. 1235–1241, 2016.

[21] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: A resilient distributed graph system on spark," in *ACM GRADES*, 2013, p. 2.

[22] J. Cipar, Q. Ho, J. K. Kim, S. Lee, G. R. Ganger, G. Gibson, K. Keeton, and E. P. Xing, "Solving the straggler problem with bounded staleness." in *ACM HotOS*, 2013.

[23] M. Li. Does MXNet support stale synchronous parallel (aka. SSP)? [Online]. Available: https://github.com/dmlc/mxnet/issues/841

[24] Apache MXNet, "https://mxnet.incubator.apache.org."

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

[26] T. Schaul, S. Zhang, and Y. LeCun, "No more pesky learning rates," in *ICML*, 2013, pp. 343–351.

[27] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Managed communication and consistency for fast data-parallel iterative analytics," in *ACM SoCC*, 2015.

[28] A. Harlap, H. Cui, W. Dai, J. Wei, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Addressing the straggler problem for iterative convergent parallel ml." in *ACM SoCC*, 2016.

[29] W. Zhang, S. Gupta, X. Lian, and J. Liu, "Staleness-aware async-SGD for distributed deep learning," in *IJCAI*, 2016.

[30] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous sgd," *arXiv preprint arXiv:1604.00981*, 2017.

[31] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *ACM EuroSys*, 2010.