FSP: Towards Flexible Synchronous Parallel Frameworks for Distributed Machine Learning

Zhigang Wang, Yilei Tu, Ning Wang, Lixin Gao, *Fellow, IEEE*, Jie Nie, Zhiqiang Wei, *Member, IEEE*, Yu Gu, and Ge Yu, *Member, IEEE*

Abstract—Myriad of machine learning (ML) algorithms refine model parameters iteratively. Existing synchronous data-parallel frameworks can accelerate training with convergence guarantees. However, the pre-assigned workload-based synchronous design still poses great challenges, since fast workers must wait for slow, straggling ones, especially in a heterogeneous computing cluster. Asynchronous alternatives can bypass this performance bottleneck, but at expense of potentially losing convergence guarantees. This article proposes a new time-based flexible synchronous parallel framework (FSP). It provides strict convergence analysis by consistently updating parameters, as well as significant cost reduction by completely unleashing the power of fast workers. It identifies the optimal synchronization frequency, by online balancing costs of parameters' update and benefits brought by their freshness. Besides the basic goal of keeping all workers fully CPU-utilized, FSP also aims to keep data spread over the cluster fully utilized, so that they can contribute to convergence with equal opportunities. These proposals are all implemented in a prototype system Flegel, with additional engineering optimizations for further performance enhancement and programming facilitation. Experiments demonstrate that Flegel significantly outperforms recent studies.

Index Terms—Machine learning, distributed computation, synchronous parallel model, straggler, workload balance

1 Introduction

Machine learning (ML) algorithms are becoming building blocks for numerous applications in the cyber world. The main idea behind ML is to iteratively train already observed data points for refining parameterized models, as much as possible, such that the output models can accurately work over other new data. Usually, an application-related objective function is defined to measure the model accuracy, and we say an algorithm converges if the function value or the number of iterations meets some pre-set criteria.

Motivation. Owing to the massive observed data volume, there is an imperative need for sound and effective ML computations. Till now, conventional efforts have made significant advancements, mainly including frequently refining model parameters for gain enhancement [1], [2], [3], [4] and parallelizing data training in a computing cluster for scalability [5], [6], [7]. All of them feature contributions in synchronous systems, where distributed workers (technically, processes or threads) follow the Bulk Synchronous Parallel (BSP) design to cooperate with each other, at a global barrier per iteration [8]. However, BSP still poses great challenges for efficiency, because fast workers must wait for slow, straggling ones. Such a negative impact is especially significant in heterogeneous environments caused by many complex factors, including not only the (static) different hardware configurations, but also the (dynamic) skewed distribution of multi-tenant workloads, usage of cheap yet transiently available computing resources [9], and non-deterministic disturbance of OS jitter and garbage collection.

We evaluate the real impact using K-means [10] over HIGGS (see Table 1) as an example. The job consists of 16 workers evenly scheduled onto 4 physical machines, followed by another with only one worker running on some machine. Then four workers of the first immediately become stragglers due to multi-tenant resource contention. Compared with the idle dedicated scenario, its runtime increases from 122secs largely up to 1050secs, leading to the 7X runtime performance degradation. On the other hand, when using transient resources, once a worker is revoked, we need to find a replacement to deploy the software and re-load data. The startup latency is roughly 1 minute even with the help of container images [11], yielding up to 50% runtime loss (and even more because revocation can happen at any time for any worker).

We are aware of many recent works on the straggler problem, but all are far from idle, like blocking data migration or proactive data replication [12], [13], [14], which requires

- *Zhigang Wang, Yilei Tu, Ning Wang, Jie Nie, and Zhiqiang Wei are with the Faculty of Information Science and Engineering, Ocean University of China, Qingdao, Shandong 266100, China. E-mail: {wangzhigang, wangning8687, niejie, weizhiqiang}@ouc.edu.cn, tuyilei@stu.ouc.edu.cn.*
- *Lixin Gao is with the Department of Electrical and Computer Engineering, University of Massachusetts Amherst, Amherst, MA 01003 USA. E-mail: lgao@ecs.umass.edu.*
- *Yu Gu and Ge Yu are with the School of Computer Science and Engineering, Northeastern University, Shenyang, Liaoning 110819, China. E-mail: {guyu, yuge}@mail.neu.edu.cn.*

Manuscript received 7 June 2022; revised 15 November 2022; accepted 8 December 2022. Date of publication 13 December 2022; date of current version 27 December 2022.
This work was supported in part by the National Natural Science Foundation of China under Grants 61902366 and U22A2068, in part by the Fundamental Research Funds for the Central Universities under Grant 202042008, in part by the National Key Research and Development Program of China under Grant 2021YFF0704000, in part by the National Natural Science Foundation of China under Grants 61902365 and 62072083, in part by National Science Foundation under Grants CNS-1815412 and CNS-1908536, and in part by the Graduate Professional Development Fund Project of Computer Department of Ocean University of China under Grant CSZS2022004.
(Corresponding author: Ning Wang.)
Recommended for acceptance by M. Si.
Digital Object Identifier no. 10.1109/TPDS.2022.3228733

TABLE 1
Datasets Summary

Algorithms	Datasets	Points ($\mathbb{P}$)	Dimensions ($\mathbb{D}$)
GMM	PUF[4]	6,000,000	128
	SUSY[5]	5,000,000	18
KMs/FCM	MASS[6]	7,000,000	27
	HIGGS[7]	11,000,000	28
LR	HIGGS	—	—
	DET[8]	6,739,534	116
NMF	ML10M[9]	69,877	10,681
	ML25M[10]	162,540	5,000
CNN	MNIST[11]	60,000	28×28, 10 labels
	SYN	120,000	28×28, 10 labels

[4] *https://archive-beta.ics.uci.edu/ml/datasets/physical+unclonable+functions*
[5] *https://archive-beta.ics.uci.edu/ml/datasets/susy*
[6] *http://archive.ics.uci.edu/ml/datasets/HEPMASS*
[7] *http://archive.ics.uci.edu/ml/datasets/HIGGS*
[8] *http://archive.ics.uci.edu/ml/datasets/detection+of+iot+botnet+attacks+n+baiot*
[9] *http://files.grouplens.org/datasets/movielens/ml-10m.zip*
[10] *http://files.grouplens.org/datasets/movielens/ml-25m.zip*
[11] *http://yann.lecun.com/exdb/mnist/*

additional network bandwidth and/or memory footprints; relaxed synchronous constraint without strict convergence guarantee, including confined synchronous [15], [16], [17], bounded stale synchronous [18], [19], [20] and asynchronous [21], [22], [23] parallel solutions, which cannot tolerate persistent (static) stragglers [24]; and tunable workload assignment customized for workers for the upcoming iteration [25], [26], [27], that still suffers from temporarily occurred (dynamic) stragglers within the current iteration.

Hence, a naturally desirable goal for ML algorithms is to pursue a new parallel model that (1) can maximally unleash the computational power of fast workers by spending time doing more useful computations instead of waiting for stragglers at the synchronous barrier; (2) can uniformly solve the static persistent and the dynamic transient straggler problem; (3) enjoys the strict convergence guarantee like BSP; and (4) is lightweight—without large additional resource requirements.

Problem Analysis. Many existing works have proved that a global synchronous barrier is essentially necessary for convergence guarantee [25], [26], [27]. However, the currently used barrier mechanism cannot cope with our first two requirements. This is because it follows a pre-defined synchronous parallel design where within each iteration, workloads measured by the number of training data points, are scheduled across workers prior to launching real computations. The synchronous barrier is passively performed if and only if every worker has already reached the pre-defined barrier location, i.e., completing pre-assigned workloads. Since workloads cannot be changed at runtime, the execution time cost is significantly sensitive to static and/or dynamic heterogeneous factors. Fast workers thereby block themselves at the barrier.

Our Contributions. In this paper, we explore a path to such a target system with a series of research and engineering efforts.

We first challenge the conventional wisdom that ML with good convergence have to be parallelized using the pre-assigned workload-based BSP framework. We now give a new time-based Flexible Synchronous Parallel (FSP) alternative. FSP still follows the design of synchronous parameter update at the global barrier, so as to ensure that all workers can always use consistent parameters for local training. That helps us formally prove the convergence. However, no workload is pre-assigned, and hence, no barrier location is pre-established. Instead, FSP enables an independent coordinator to actively initiate a barrier to synchronize all workers whenever necessary. Once receiving such a notification, a worker will pause right after atomically training a single data point, and then immediately commit local intermediate results to refine global parameters. Since the cost of an atomic operation is tiny, all workers can pause at nearly the same time instance, even some of which are persistent or transient stragglers.

Pioneering studies reveal that frequently initiating barriers can keep parameters fresh and hence improve training quality [1], [4]. While, although FSP removes waiting time, its synchronization is still not free. During the iterative training, dynamically identifying a proper synchronization frequency (interval) can strike a good balance between costs and benefits. The flexibility of FSP, however, forces programmers to either blindly select an interval or experience a long learning curve to understand the internals of underlying engines. Existing dynamic barrier efforts (namely batch size in BSP) [2], [3] cannot work very well in FSP, since they ignore the impact of stragglers on runtime. To gain overall success while insulating programmers from the tedious low-level details, we design a multi-stage adjusting component to adaptively seek optimal intervals. Within one stage, we linearly simulate the changing trend of the objective function value, for recommending an interval, and hold it steady to avoid over-reaction or oscillation. The fitting error can be self-corrected by dividing training process into multiple stages and then re-performing recommendation.

Typically, training data are evenly distributed across the computing cluster, as workers' runtime states are usually unavailable before iterations. Together with the fact that FSP keeps all workers fully CPU-utilized, data on fast workers are inevitably traversed more times than on stragglers, yielding biased influence on parameters and even making the ML model overfit to the former. Excessive traversing also weakens the average convergence contribution of a single pass, leading to the low cost-benefit ratio. We solve this data under-utilization problem by a traditional yet effective manner, i.e., dynamically migrating data from stragglers to fast workers. But the differences against prior load balance works [12], [14] are twofold. First, our goal is, in remaining iterations, to make migrated lazy data catch up with excessive traversed data, in terms of the traversing times, rather than equalizing the runtime of workers (has already guaranteed by the basic FSP). We thereby design totally different policies to smartly select data and decide migration timing, to dynamically compensate the data on stragglers for their missing traversal times. Second, since FSP has decoupled barriers and local training workloads, we can give a bi-directional scheduling policy to overlap migration streaming and local training streaming. Data thereby can be migrated asynchronously across iterations to completely hide the migration cost.

In a nutshell, we make the following contributions.

- *Proactive flexible synchronous parallel framework FSP*, which keeps workers fully CPU-utilized by time-

based barriers, to handle persistent and transient stragglers, and guarantees convergence by consistent parameter update.
- *Adaptive synchronization interval optimization*. By multi-stage self-corrected linear fit, its robustness adjustor can quickly identify such a hyper-parameter and smartly react to stragglers, without any runtime-sensitive input.
- *Lightweight dynamic workload balance* with a new goal of keeping all data evenly utilized and hence equally contributing to convergence. It asynchronously migrates data to avoid blocking local training.
- *Prototype system Flegel*. It exposes uniform APIs and multiple convergence criterions to end-users, for easily programming various ML algorithms. Because stragglers also react to the coordinator slowly, *Flegel* gives a semi-centralized coordination design to mitigate the delay impact, by allowing a straggler to actively react in advance.

We state that the FSP scheme has been introduced in our early work [28], but only for partial ML algorithms. Now we generalize it to common ML computations. Besides, although the early built-in binary searching based adjustor and its variant [24], [28] can give an interval recommendation by multiple attempts, we now propose a multi-stage linear fit alternative with strong robustness and the capability of quickly seeking optimization. We particularly extend FSP by adding light-weight load balance with new metrics and migration techniques. Engineering efforts like semi-centralized coordination and multiple convergence criterions are also newly proposed. All experiments are re-organized and re-tested, including more ML algorithms on a broad spectrum of real datasets, to show the generality and advantage of *Flegel*.

Paper Organization. Below, Section 2 gives a background introduction. Section 3 presents FSP with interval seeking optimization. Section 4 introduces dynamic load balance. Section 5 discusses the *Flegel* system. Section 6 reports evaluation results. Section 7 highlights related works and Section 8 finally concludes this work.

2 Preliminaries

This section briefly reviews ML training strategies. Without loss of generality, we use Gradient Descent (GD) [29] and Expectation Maximization (EM) [30] as representatives to demonstrate our contributions, but others like Coordinate Descent [31], Model Average [32], BUMF [33] and ADMM [34], are also applicable.

Let X be an observed value of some random variable, typically consisting of n independent data points. X can be decomposed as $\{X_1, \ldots, X_n\}$. The general goal of ML is to refine a model with parameter θ by iterating over input data X, so as to perform future predictions over new data. The refining quality (model accuracy) is measured by the objective function $f(\theta)$.

2.1 Gradient Descent

Gradient Descent (GD) is the most widely used training strategy. It computes function-derivative-based gradients by training data, in order to give the most-right value-descent direction for $f(\theta)$. Here, $f(\theta)=\frac{1}{n}\sum_{i=1}^{n}F_i(X_i;\theta)$, and $\forall X_i \in X$, $F_i(X_i;\theta)$ indicates the loss w.r.t. X_i. GD computations consist of two procedures at each iteration, i.e., computing gradient ∇F_i and then updating parameter θ. The former can be a stochastic gradient trained by a single data point (SGD) or the average by the whole (Full-batch). Eq. (1) gives another Mini-batch choice [4] to refine parameter $\theta^{(t)}$ at the t-th iteration. It samples a few of points between two consecutive iterations/barriers to form the mini-batch set B for averaging gradients. That is a better compromise between noise incurred by SGD, and the inefficiency of broadcasting fresh θ in full-batch GD. Here α is a hyper-parameter *learning rate*, which affects the model accuracy and is beyond the scope of this paper

$$\theta^{(t)} = \theta^{(t-1)} - \alpha \frac{1}{|B|} \sum_{X_i \in B} \nabla F_i. \tag{1}$$

Logistic Regression (LR). We use LR to show how mini-batch GD trains its model parameter θ. Each data point $X_i \in X$ contains r attributes with an additional boolean ground-truth value y_i, that is, $X_i=\{X_i, y_i\}=\{\{x_{i1}, x_{i2}, \ldots, x_{ir}\}, y_i\}$. LR on X returns a function χ, which predicts $y_i=1$ based on θ with probability

$$\chi(X_i) = \frac{1}{1+exp(-X_i^T\theta)}.$$

The optimal θ minimizes the loss function

$$F(X_i;\theta) = (y_i - 1)\log(1-\chi(X_i)) - y_i \log \chi(X_i),$$

and each dimension $\theta_k^{(t)}$ of θ is refined by the related gradient

$$\theta_k^{(t)} = \theta_k^{(t-1)} - \alpha \frac{1}{|B|} \sum_{X_i \in B} \frac{\partial}{\partial \theta_k} F(X_i; \theta^{(t-1)}), 1 \le k \le r.$$

2.2 Expectation Maximization

As one of the top 10 data mining algorithms, Expectation Maximization (EM) can effectively train models with an additional unobserved hidden variable Z. Z is associated with X, and the range domain of each $Z_i \in Z$ is $\{z_1, \ldots, z_l\}$. Now we wish to find a maximum log likelihood estimate $f(\theta)$ for parameter θ.

EM solves this problem by alternately estimating expectation of Z (called E-step) and maximizing marginal likelihood for $f(\theta)$ (called M-step), resembling the two procedures in GD. At the t-th iteration, given the current parameter $\theta^{(t-1)}$, E-step estimates a distribution $Q_i^{(t)}$ over the range of Z_i w.r.t. every X_i (full-batch), i.e., $Q_i^{(t)}(z_k)=P(z_k|X_i,\theta^{(t-1)})$, s.t. $\sum_{z_k} Q_i^{(t)}(z_k)=1$ and $Q_i^{(t)}(z_k)\ge 0$. Then M-step updates $\theta^{(t)}$ to the θ that can maximize $f(\theta)=\sum_{i=1}^{n} E_{Q_i^{(t)}}[\log P(z_k, X_i|\theta)]$, where $P(z_k, X_i|\theta)$ indicates the joint probability for Z_i and X_i based on θ, and $E_{Q_i^{(t)}}[\cdot]$ denotes expectation w.r.t. $Q_i^{(t)}$ found in E-step.

A mini-batch variant can also achieve prominent performance, through training data from a subset $B \subseteq X$ in E-step [1], [10]. That generates two modifications. On one hand, θ^t is updated based on a new statistics vector $s^{(t)}=\sum_{i=1}^{n} s_i^{(t)}(Z_i, X_i)$, where $s_i^{(t)}(Z_i, X_i)$ is associated with

$Q_i^{(t)}(z_k)$ and can be easily computed the change Δs_i. As a result, even partial data are computed in E-step, by accumulating related Δs_i, we can incrementally update $s^{(t)}$ and hence correctly refine θ. On the other hand, the training goal is equivalently re-defined as that both the E and the M steps try to maximize, or at least increase a new objective function shown in Eq. (2) [1], where $H(\cdot)$ is the entropy of Q_i

$$F(Q,\theta) = \sum_{i=1}^{n} F_i(Q_i,\theta), where$$
$$F_i(Q_i,\theta) = E_{Q_i}[\log P(z_i, x_i|\theta)] + H(Q_i). \quad (2)$$

Eq. (3) finally shows the computation flow of mini-batch EM, at the t-th iteration. Here, $Q_i^{(t)}$ is set to the Q_i that maximizes $F_i(Q_i,\theta)$, given by $Q_i^{(t)}(z_k) = P(z_k|X_i,\theta^{(t-1)})$.

$$\begin{cases} \textbf{E}-\textbf{step : Choose } B \textit{ to be updated, and } \forall X_i \in B: \\ \textbf{Set } s_i^{(t)}(Z_i,X_i) = E_{Q_i^{(t)}}[s_i^{(t)}(Z_i,X_i)]. \\ \textbf{Set } \Delta s_i^{(t)} = s_i^{(t)}(Z_i,X_i) - s_i^{(t-1)}(Z_i,X_i). \\ \textbf{Commit } every\ \Delta s_i^{(t)},\ X_i \in B. \\ \textbf{Wait } for\ newly\ updated\ \theta^{(t)}. \\ \textbf{M}-\textbf{step : Set } s^{(t)} = s^{(t-1)} + \Delta s_i^{(t)},\ X_i \in B. \\ \textbf{Set } \theta^{(t)}\ to\ the\ \theta\ that\ maximizes\ F(Q,\theta) \\ based\ on\ s^{(t)}. \\ \textbf{Broadcast } \theta^{(t)}. \end{cases} \quad (3)$$

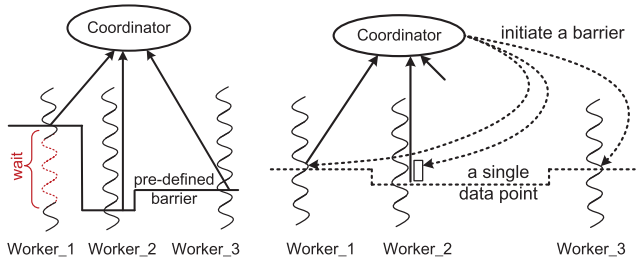
K-Means Example. As a simple EM application in clustering, K-means aims to partition n observed data points in X into l clusters $\theta = \{\theta_1, \ldots, \theta_l\}$ so as to minimize $f(\theta) = \sum_{j=1}^{l} \sum_{X_i \in \theta_j} ||X_i - \mu_{\theta_j}||$, where $\mu_{\theta_j} = \frac{1}{|\theta_j|} \sum_{X_i \in \theta_j} X_i$ is the centroid of θ_j. The range of Z_i is $\{1, 2, \ldots, l\}$, indicating to which of l clusters a given observed X_i is supposed to be assigned. E-step assigns X_i to the nearest cluster $\theta_{\tilde{j}}$ for $k \in [1, l]$: $Q_i^{(t)}(z_k) = 1$, if $z_k = \tilde{j}$; and 0 otherwise. The statistics $s^{(t)}$ includes two vectors with l-dimensions: S with $S_j = \sum_{X_i \in \theta_j} X_i$, and C with $C_j = |\theta_j|$. Suppose that X_i is moved from θ_j to $\theta_{j'}$. S and C are updated incrementally by: $S_j = S_j - X_i$, $S_{j'} = S_{j'} + X_i$; $C_j = C_j - 1$, $C_{j'} = C_{j'} + 1$. In M-step, θ_j is updated by $\mu_{\theta_j} = \frac{S_j}{C_j}$.

2.3 Distributed Machine Learning

For better scalability, myriad of today's ML systems employ the underlying *parameter-server* framework [22] where data points X are distributed across multiple workers for parallel training (also called *data-parallelism*), and the model parameter θ is kept on logical servers (physically may also be data workers). At the global barrier, gradients in GD or incremental changes in EM are aggregated from workers to servers, to refine parameters. Such a design ensures that all workers can see consistent parameters, which provides convergence guarantee. But it suffers from expensive waiting costs caused by stragglers, —which our new flexible synchronous parallel framework can reduce.

3 Flexible Synchronous Parallel

We now introduce our Flexible Synchronous Parallel (FSP) framework. We first present its overview design (Section 3.1),



(a) Pre-defined barrier (b) Flexible barrier

Fig. 1. Illustration of performing a global barrier under (a) existing BSP and (b) new FSP. The dashed line in (b) indicates a flexible barrier. Besides, the solid arrow line in (a) and (b) indicates committing local training results, like ∇F in GD and Δs in EM, to the coordinator.

followed by a theoretical guarantee on convergence (Section 3.2) and how to seek an optimal synchronization interval (Section 3.3).

3.1 Overview Design

FSP is designed to reduce the waiting time in traditional pre-defined bulk synchronous parallel (BSP) frameworks. Under FSP, fast workers can perform more useful computations to accelerate convergence, while the strict convergence feature as provided by BSP, can still be guaranteed.

The waiting time in BSP is mainly caused by its built-in pre-defined barriers. Before running a ML iteration under BSP, all workers pre-define barrier locations by assigning workloads, like processing every local data point (full-batch) or some block B (mini-batch). A global barrier is passively formed when all workers have completed such workloads, which is simple but very sensitive to stragglers. Suppose that three workers are used. As shown in Fig. 1a, they possibly reach their individual pre-defined barriers at different speeds. The coordinator cannot update parameters or broadcast their new version, until it receives all local training results. Fast workers like *Worker_1* thereby block themselves to wait for stragglers like *Worker_2*.

FSP replaces the pre-defined barrier with a new flexible mechanism. Now the coordinator can actively initiate a barrier by broadcasting a signal. Once receiving the signal, a worker immediately commits local training results right after computing the current data point, no matter how many have been processed since the last barrier. Data points processed at the t-th iteration are encapsulated into the subset/mini-batch $\tilde{B}^{(t)}$. Its size clearly indicates the workloads per iteration. Different from the pre-defined B in Eqs. (1) and (3) under BSP, $\tilde{B}^{(t)}$ varies with iterations, because a new flexible barrier can be initiated at any time. As shown in Fig. 1b, workers first committing results, such as *Worker_1*, only wait for stragglers, such as *Worker_2*, to process a single data point at most. Local results are abstract summations collected from individually processed data points, like $\sum \Delta F_i$ in GD and $\sum \Delta s_i$ in EM, which can be maintained in an incremental manner. Hence, the coordinator can quickly complete a synchronization operation. As a result, fast workers spend more time performing computations, instead of waiting for stragglers. Also, the lightweight synchronization barrier can be initiated more frequently so that more fresh parameters are visible for workers. Two advantages contribute to boosting the performance.

On the other hand, each worker under FSP uses a round-robin policy to schedule local training operations among iterations, so that every data point can be evenly sampled during computations. Like BSP, here workers also share the same parameters. This is because the coordinator will not refine parameters until all workers have already committed their local results at the barrier. Training on workers, in return, can continue only when receiving new parameters from the coordinator. The consistent parameter view guarantees convergence (see Section 3.2).

Last but not least, assume that a worker begins to process a single data point right before receiving the barrier signal. It can make a smart decision about whether or not to continue the local training, so as to timely respond to the signal. In most cases, computing one data point leads to negligible waiting costs. It is acceptable that the worker participates in synchronization after completing the local operation. However, in some real-world applications, processing a single data point is very time-consuming. For example, ML algorithms in the global atmospheric model, need to extract features from high-resolution images. Such images usually have much more pixels than those used in regular classification tasks (e.g., 3600×2400 in Ref. [35] versus 28×28 in Table 1 in our experiments), which largely increases the time complexity of processing a single image. Now the worker should immediately abandon the operation. Otherwise, other workers need to wait for a long while, which wastes compute resources.

3.2 Convergence Analysis

The key advantage of FSP is that it can guarantee ML convergence. We show the formal analysis in Theorems 1 and 2, respectively for the representative GD and EM training strategies.

Theorem 1. *GD-based ML algorithms under FSP converge.*

Proof. Traditionally, the convergence of GD is analyzed with assumptions that the objective function f is continuously differentiable and c-strong convexity, and the gradient function ∇F is L-Lipschitz continuous. We then utilize existing analysis for proof, based on features of FSP.

Recall that the mini-batch size $|\tilde{B}|$ under FSP dynamically varies with iterations, as a barrier can be arbitrarily issued. Léon Bottou et al. prove that the convergence is insensitive to $|\tilde{B}|$ [36]. They show the expected optimality gap between $f(\theta^t)$ and the optimal $f(\theta^*)$ can be bounded by the learning rate α and the constant scalars c and L (see Theorem 5.1), all of which are technically orthogonal to FSP. On the other hand, FSP guarantees that all workers can see consistent parameters, which eliminates any possible error in the bounded optimality gap [18]. □

Theorem 2. *EM-based ML algorithms under FSP converge.*

Proof. Convergence guarantee can be proved by showing that EM computations monotonously increase the objective function value $F(Q,\theta)$ (Eq. (2)). Towards this end, we prove that each EM iteration consisting of E-step and M-step, either increases $F(Q,\theta)$ or leaves it unchanged.

At the m-th iteration from the coordinator view, M-step maximizes $F(Q^{(m)},\theta^{(m)})=\sum_{i=1}^{n}F_i(Q_i^{(m)},\theta^{(m)})$ through changing $\theta^{(m-1)}$ to $\theta^{(m)}$. This update is based on the global statistics s derived from the change of Q. In particular, if X_i is not processed, we assume that its Q_i is left unchanged (Δs_i=0). Obviously, M-step can monotonously increase $F(Q,\theta)$.

From the j-th worker view, at the t_j-th iteration, E-step updates $F_i(Q_i^{(t_j+1)},\theta^{(t_j)})$, and hence $F(Q,\theta)$, by changing $Q_i^{(t_j)}$ to $Q_i^{(t_j+1)}$. To prove E-step continuously increases $F(Q,\theta)$, we shall prove that the newly updated F_i is greater than, or at least equal to the value used in the last M-step, as shown in Eq. (4).

$$F_i(Q_i^{(t_j+1)},\theta^{(t_j)}) \geq F_i(Q_i^{(m)},\theta^{(m)}) \tag{4}$$

Based on the assumption in M-step, $Q_i^{(t_j)} \equiv Q_i^{(m)}$. We then re-write Eq. (4) in the following:

$$F_i(Q_i^{(m+1)},\theta^{(t_j)}) \geq F_i(Q_i^{(m)},\theta^{(m)}) \tag{5}$$

On the other hand, conditioned on $\theta^{(t_j)}$, we can maximize $F_i(Q_i^{(m+1)},\theta^{(t_j)})$ using a Lagrange multiplier [1], subject to $\sum_{z_k}Q_i^{(m+1)}(z_k)=1$ and $Q_i^{(m+1)}(z_k)\geq 0$. At such a maximum, we have the unique solution that $\hat{Q}_i^{(m+1)}(z_k)=P(z_k|X_i,\theta^{(t_j)})$, and we indeed use it to initialize $Q_i^{(m+1)}(z_k)$. Therefore,

$$F_i(Q_i^{(m+1)},\theta^{(t_j)}) \geq F_i(Q_i^{(m)},\theta^{(t_j)}) \tag{6}$$

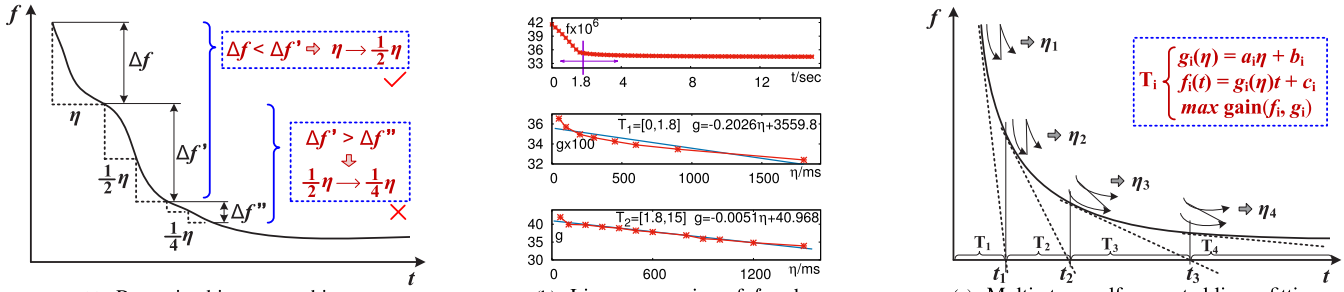
Because no worker can continue local computation before receiving the new parameter θ, we can easily infer that $\theta^{(t_j)}=\theta^{(m)}$. Based on Eq. (6), we can establish the correctness of Eq. (5), and hence E-step also increases $F(Q,\theta)$. Together, $F(Q,\theta)$ is monotonously increased as desired. □

Neal et al. [1] have proved the strict convergence property, i.e., monotonously increasing $F(Q,\theta)$ at each iteration, for centralized EM algorithms. However, their proof ignores the discussion on different versions of Q_i and θ, because the sequential model naturally ensures that subsequent operations can immediately see any update result. Differently, Theorem 2 tells us that a global barrier is essentially required in distributed environments. Otherwise, the comparison result between $F_i(Q_i^{(m)},\theta^{(m)})$ in Eq. (5) and $F_i(Q_i^{(m)},\theta^{(t_j)})$ in Eq. (6) is non-deterministic. This is because M-step focuses on maximizing the overall summation $F(Q,\theta)$, instead of a single $F_i(Q_i,\theta)$. Our FSP framework is thereby different from relaxed synchronous frameworks [15], [18], [22].

3.3 Optimal Synchronous Interval

Although a flexible barrier can be initiated at any time without compromising the convergence guarantee, the question is that who can trigger it and when? Now we answer the two key issues.

As an extreme case, given a big synchronous interval η, it is most likely that a worker has cycled through local data before receiving a barrier signal. Since θ is left unchanged



(a) Recursive binary searching (b) Linear regression of f and g (c) Multi-stage self-corrected linear fitting

Fig. 2. Optimal barrier interval under FSP.

within an iteration, computations in multiple full data traversals are redundant, except the first. Therefore, to avoid wasting compute resources, any worker already completed a traversal pass, will immediately notify the coordinator for synchronization.

Pioneering experience, however, tells us that frequent synchronization with a relatively small η, can make up-to-date parameters visible, and then improve the subsequent training quality, namely, making great progress for optimizing $f(\theta)$ per iteration [1], [4]. While, the flexible barrier is still not free, owing to network costs and the parameter refinement delay. Such an η setting inevitably increases synchronization costs, which decreases the quantity, i.e., iterations executed within unit time. Clearly, it is very necessary for coordinator to identify a proper interval, to hit a "sweet spot" between quality and quantity.

Overall, the coordinator should actively play the trigger role, and transfer the power to workers if η is too big. Next, our focus is how to compute η. The key is how to fully utilize online statistics and offline knowledge. Below, we first outline our early binary search based solution as a baseline, and then give details of the new multi-stage fitting design.

Recursive Binary Searching. The main idea is to compare the change of the objective function value $f(\theta)$ within unit time under the current interval η and a tentative smaller $\frac{\eta}{\lambda}$, respectively. Here λ is the user-specified adjusting factor and typically set to 2. As shown in Fig. 2a, $\eta = \frac{\eta}{\lambda}$ takes effect if the change Δf w.r.t. η after some iteration, is smaller than $\Delta f'$ w.r.t. $\frac{\eta}{2}$, which is accumulated across the strictly subsequent λ iterations. This is because the comparison result implies that benefits from fine-grained synchronization outperform costs of additional $(\lambda - 1)$ barriers, i.e., quality beats quantity. The adjusting operation is recursively continued until the result is reversed, like $\Delta f' > \Delta f''$. Prior experiments [37] reveal that the batch size that yields the best speedup during initial iterations is roughly the optimal in the whole. Hence, we also fix η unless the environment changes.

Note that Δf and $\Delta f'$ are computed from different training phases. To ensure a fair comparison and then analyze the accurate impact of adjustment, our binary searching assumes that $f(\theta)$ strictly linearly varies with elapsed time. However, such a strong assumption usually cannot hold true in practice. The variation speed gradually degrades as iterations proceed, because the model trends towards stability. The natural degradation inevitably weakens the training quality, although we collect Δf and $\Delta f'$ in adjacent iterations. That generates a false positive comparison result and hence early terminates adjustment. Based on our tests, binary searching only works at the very beginning of iterations, because now the degradation is less significant. In addition, it needs $\log \eta^0/\eta^*$ adjustments from the initial input η^0 to the optimal η^*. The complexity is so sensitive to η^0 that the valid adjusting window period is further shortened, if a big input is given. Our early version uses a full-batch runtime as η^0; a variant optimizes its value with literature researches [24], both of which prolong η^0 when stragglers suddenly happen.

Multi-Stage Self-Corrected Linear Fitting. We now propose a new robustness component to seek η^*, with relaxed linear assumption and input sensitivity.

The key insight is, by extensive tests on many ML algorithms, although f against elapsed time t does not follow a linear trend in the entire training, the fitting error is largely reduced in a short period. The top subfigure in Fig. 2b demonstrates K-means as an example. An inflection point is clearly observed at $t = 1.8$secs. The binary searching can be easily terminated around this point, because of the sharp degradation of training quality. However, observations in the two separated stages $T_1 = \{0, 1.8\}$ and $T_2 = \{1.8, 15\}$, respectively follow a linear regression model. That motivates us to seek η^* separately. To further relax the input sensitivity, we also replace the time-consuming recursive searching with a fast cost-benefit estimation, as mathematically shown in Eq. (7). The main idea behind it is to maximize the optimization gain w.r.t. f in the valid training time, by seeking the optimal η^*. Let φ stand for the barrier cost. In some linear fitting stage T, the valid time is computed by excluding the costs of total $\frac{T}{\eta+\varphi}$ barriers. Here, $g(\eta)$ is a quality function indicating the training progress within unit time. Fortunately, as shown in middle and bottom subfigures in Fig. 2b, g also linearly varies with η within each stage. Till now, as described in Fig. 2c, we can compute an optimal interval η_i^* for each stage T_i

$$\max\ gain(\eta) = \left(T - \varphi \cdot \frac{T}{\eta + \varphi}\right) \cdot g(\eta). \tag{7}$$

The next problem is how to reasonably divide stages. Ideally, the total training time is predictable if the linear regression of f on t has $R^2 = 1$. Given the current objective value f_{cur} and the target f_{tar}, it is $(|f_{cur} - f_{tar}|)/g(\eta)$. However, under multi-stage-division, the fitting error is just reduced but still not zero. The accumulated errors across iterations yield convergence delay, namely, the ML algorithm does not converge as predicted. As shown in Fig. 2c,

once such a delay is detected, we should enter into the next stage and recompute another η^* to correct fitting errors. By continuously monitoring the training progress, we can adaptively divide stages and hence self-correct errors.

We finally detail the estimation of $g_i(\eta)$ within the i-th stage T_i. Its linear fitting $g_i(\eta)=a_i\eta+b_i$ includes two hyper-parameters, which requires two points $(\frac{\Delta f_i(\eta_i)}{\Delta t_i}, \eta_i)$ and $(\frac{\Delta f_{i+1}(\eta_{i+1})}{\Delta t_{i+1}}, \eta_{i+1})$ with different interval attempts for estimation. Given η_i, through continuously running two iterations starting from X_j, $\frac{\Delta f_i(\eta_i)}{\Delta t_i}$ indicates the training quality during the valid elapsed time $\Delta t_i=2\eta_i$.[1] We then rollback to the original starting location X_j and train the same data again with $\eta_{i+1}=2\eta_i$ to get another point. Such a design clearly overlaps training data used in estimation, and hence eliminates any possibly skewed impact. Because η_i is usually very small, we can tolerate the cost of redundant computations.

4 Lightweight Dynamic Workload Balance

The basic FSP in Section 3 has guaranteed full utilization of CPU resources. However, it might generate and exacerbate the skewness of data traversals across workers, and finally slow down convergence speed. Our solution is to dynamically balance workloads, which has been well studied but now we have a new goal and completely different techniques.

4.1 From Idle CPU to Idle Data

Before training, distributed ML often shuffles data points among total ω workers by random hash, so as to follow the Independently Identical Distribution (IID) assumption. On the other hand, owing to pre-assigned workloads, when ML converges, BSP ensures data D_i on each worker *Worker_i* (W_i for short) can be traversed exactly the same number of passes p_i^*, $1\leq i\leq\omega$. Both of the two conditions guarantee that workers can commit roughly equally-important local results. For example, $\forall i\neq j$, we can have the summation of gradients in GD

$$\sum_{k=1}^{p_i^*}\sum_{X_h\in D_i}\Delta F_h^k\approx\sum_{k=1}^{p_j^*}\sum_{X_h\in D_j}\Delta F_h^k,$$

and accumulated changes in EM

$$\sum_{k=1}^{p_i^*}\sum_{X_h\in D_i}\Delta s_h^k\approx\sum_{k=1}^{p_j^*}\sum_{X_h\in D_j}\Delta s_h^k.$$

That is to say, they evenly contribute to parameter refinement. However, as analyzed in Section 3.1, fast workers are blocked at the pre-established barriers, and hence their CPUs are idle.

Differently, workers under FSP make the best-efforts to maximize their individual traversal passes. p_i^* of fast worker W_i is then greater than p_j^* of straggler W_j. The quantity gap clearly destroys the second condition. Meanwhile, model parameters inevitably overfit to data D_i, as they are trained too often; that in turn reduces the quality like ΔF_h^k or Δs_h^k from a single data point in D_i. Compared against BSP, FSP of course can accelerate convergence because W_i does not waste its compute power and then contributes more results. However, the skewness in quantity and quality is newly incurred as mentioned above, because of the relatively idle data on stragglers (traversed with fewer times). In another word, FSP shifts utilization skewness from CPU to data. That opens up a new space for further performance improvement.

Now we argue that the additional gain can be achieved, if fast worker W_i uses its extra power to train fewer-traversed data D_j on straggler W_j, rather than its own D_i. This is because quality from D_j is more significant. A straightforward implementation is data migration. But different from traditional works focusing on equalizing runtime in only remaining iterations, now we aim to keep data equally-utilized measured by $p_i^*\approx p_j^*$ $(i\neq j)$, in the whole training process. That means we should not only balance future passes, but also compensate previously accumulated loss. We then require a new fine-grained balance policy as described in Section 4.2. Further, FSP decouples barrier locations and local workloads, that enables us to flexibly migrate data in an asynchronous manner, and even across iterations, as shown in Section 4.3. This is different from existing blocking policies where migration only happens within a single iteration or at the barrier.

4.2 Fine-Grained Workload Balance

Before showing the detailed solution, we first formulate our new balancing goal in Eq. (8). For simplification and efficiency, we real-timely group homogeneous workers $\{W_{i1},\ldots,W_{ik},\ldots\}$ with the same statistics into a super worker $\mathbb{W}_i$. The statistics include the number of currently maintained data points n_{ik}, training speed v_{ik}, and the already completed traversal passes p_{ik}. $\mathbb{W}_i$ has the same corresponding statistics N_i, V_i, and P_i. When ML converges, we say the training process is balanced if the difference between the specific passes of any super worker and the average of total K super workers, does not exceed a given threshold ϵ.

$$\left|P_i^*-\frac{\sum_{j=1}^{K}P_j^*}{K}\right|\leq\epsilon, 0\leq\epsilon\leq 1 \tag{8}$$

We then continuously monitor P_i at each barrier and start data migration once the threshold is exceeded. Now the question is migrating how many data from selected sources to targets. Note that we can estimate the remaining time T with the same method used in Section 3.3, and then the average passes $\frac{1}{K}\sum_{j=1}^{K}P_j^*$ until convergence. Both help us to carefully calculate how many data should be migrated into/out-from workers in every $\mathbb{W}_i$, so that P_i and P_j are definitely roughly equal, for any $i\neq j$. However, the predicted T might be inaccurate because of the non-linear variation of $f(\theta)$. Worse, our asynchronous across-iteration migration (shown in Section 4.3) cannot guarantee that data on stragglers immediately enjoy the power of fast workers. We even do not know when migration is completely done.

1. η_i is user-specified and set to 200 ms by default.

To mitigate these problems, we propose the fine-grained balance design with a real-time adjusting function. It evenly splits T into d parts and within each one, a peer-to-peer migration policy is used to balance passes of paired super workers. This gradually reduces the overall difference with d rounds of matching, which provides opportunities to adjust paired candidates across rounds, to cope with the inaccurate prediction and unblocking migration.

Specifically, before starting the k-th round, we predict passes $\tilde{P}_i^k = P_i^{k-1} + \frac{T}{dV_i^{k-1}N_i^{k-1}}$, assuming no data is migrated in this $\frac{1}{d}T$ time split ($N_i^0 = n_{i1}$ and $P_i^0 = p_{i1}$ for $k=1$). We sort the total K super workers in ascending order of $\tilde{P}_i^k$. Our matching rule is simply selecting the minimum and the maximum as a pair, and the rest can be deduced in the same manner. Taking $< \mathbb{W}_i, \mathbb{W}_j >$ as an example, to equalize P_i^k and P_j^k, partial data are evenly collected from workers in $\mathbb{W}_i$, and then evenly assigned to workers in $\mathbb{W}_j$. Note that $\mathbb{W}_i$ and $\mathbb{W}_j$ might have different numbers of workers. If a regular slow worker contributes Δ_{ij}^k data points, then the corresponding fast worker will receive $\beta\Delta_{ij}^k$ where $\beta = \frac{\mathbb{W}_i}{\mathbb{W}_j}$. Eq. (9) mathematically shows the calculation of Δ_{ij}^k. The idea behind it is that migration can accelerate the process of remaining data for every worker in $\mathbb{W}_i$, while simultaneously reducing the speed for $\mathbb{W}_j$ since more data need to be traversed. In particular, $(P_j^{k-1} - P_i^{k-1})$ indicates the lost passes of migrated data. We should exclude the compensation time from $\frac{1}{d}T$ when estimating the newly added passes in $\mathbb{W}_j$

$$P_i^{k-1} + \frac{T/d}{(N_i^{k-1} - \Delta_{ij}^k)V_i^{k-1}} = P_j^{k-1} + \frac{T/d - (P_j^{k-1} - P_i^{k-1})\beta\Delta_{ij}^k V_j^{k-1}}{(N_j^{k-1} + \beta\Delta_{ij}^k)V_j^{k-1}}. \quad (9)$$

A full balancing cycle cannot terminate until $k \geq d$ or Eq. (8) is met. For $k \geq d$, another cycle is launched if ML has not converged. Another problem is that a time split might even not be enough for compensation, i.e.,

$$T/d - (P_j^{k-1} - P_i^{k-1})\beta\Delta_{ij}^k V_j^{k-1}.$$

Then the two paired super workers should work together for compensation. Note that now the goal of migration is to equalize the compensation time respectively for remaining data and migrated data. The Δ_{ij}^k is then computed by the following equation:

$$(N_i^{k-1} - \Delta_{ij}^k)V_i^{k-1} = \beta\Delta_{ij}^k V_j^{k-1}.$$

4.3 Asynchronous Data Migration

An idle asynchronous policy can guarantee that, the local training streaming is not at all aware of data migration, for every worker including the paired source and target. We decompose this goal into two parts: completely hiding communication costs and avoiding any possible loss of traversal passes for data flying on network. The former can be naturally achieved. This is because FSP does not care about how many data points are processed by a specific worker, which enables flexible data exchange. But for the latter, we should very carefully manage migrated data at both source and target sides. Otherwise, they may be skipped by normal traversals.

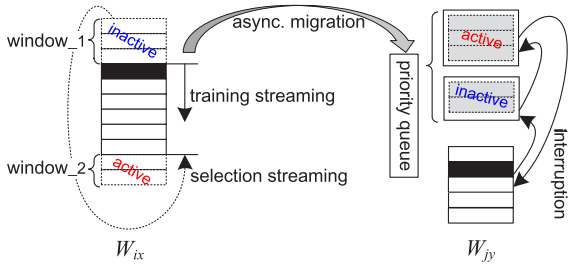


Fig. 3. Overlap of local training and migrated data selection. Solid and dashed blocks respectively indicate native data and selected migrated data. The black block indicates the current computation focus. $W_{ix} \in \mathbb{W}_i$ and $W_{jy} \in \mathbb{W}_j$ are respectively regular slow and fast workers.

Our management solution consists of a bi-directional sliding window method at source and an on-demand interruption policy at target. As shown in Fig. 3, the training streaming at the source side rotates local data in a top-down direction. The most recently trained data will not be computed again until a full-batch is done. Owing to the traversal-safety, we continuously select the total Δ_{ij}^k pieces of local data points for migration, in the reversely bottom-up direction right before the current computation focus. Note that migration might continue fitfully over a quite long period, because of resource contention. We then partition these data into different windows. Only data in the current window, not all, will be locked for sending and hence prevented from local training. Once they are successfully sent, we slide to the next window. On the other hand, data within a window will be received by the target as a whole and put into a queue. Within an iteration, the target always trains newly migrated data with high priority and then continues over locally native data. Such an interruption makes the former catch up with the latter as quickly as possible, from the perspective of traversal passes. Data are removed from the priority queue and regarded as native when their lost passes have already been compensated. Till now, we say FSP can smoothly embrace data migration without large loss penalty of passes.

Recall that the estimated remaining time T is typically smaller than its true, owing to the gradually degraded training quality. By Eq. (9), more data are migrated to balance the pass numbers in such a short period. If ML does not converge as estimated, then original stragglers become "fast" because the fewer remaining data on them will be over-traversed. Now we should perform migration in reverse direction but clearly waste extra network bandwidth. We solve this problem by virtual migration. In another word, migrated data will not be physically removed from source workers. Instead, they are still preserved but marked as "inactive" to escape from local training. Once upon required, we shift the logic computation task by marking partial of them, like *window_2*, as "active" again.

5 Flegel: A Flexible ML System

Now we present *Flegel*, an open-source memory-based distributed system atop our FSP framework[2]. It particularly

2. https://github.com/FSPML/FSPML

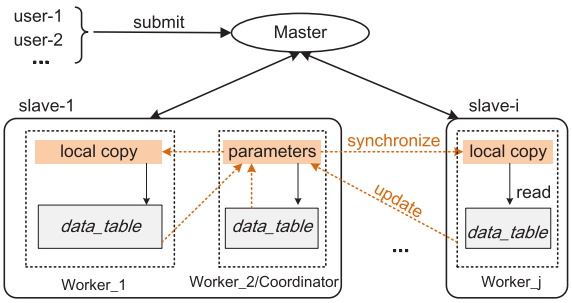


Fig. 4. Architecture of *Flegel*.

```
// initialize parameters
P initParameter( );
// compute one data point
A updateDataPoint(D dataPoint, P parameter, int iterationNum);
// accumulate statistics got from updateDataPoint()
A aggregate(A target, A aggregator);
// update parameters
P updateParameter(A aggregator, P oldParameter);
// terminate iterations or not
boolean terminationCheck(A aggregator, int iterationNum);
// stop training a data point
void voteToHalt( );
```

Fig. 5. APIs provided by *Flegel*.

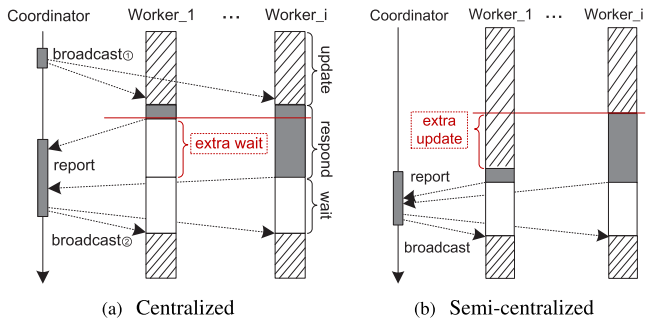


Fig. 6. Different coordination policies under FSP.

implements an efficient barrier coordination design, and provides different convergence termination check mechanisms.

5.1 Parameter Server Framework

Architecture. Fig. 4 gives the overview architecture of *Flegel*, which employs a widely used *parameter server* framework with the underlying Master-Slave design in cloud computing. Master is in charge of slaves, including monitoring their healthy status and being aware of the load variation. It also responds to concurrent requests of running ML algorithms submitted by multiple users.

The execution of one job is divided into several workers which are scheduled across slaves and host different partitions of input data to support *data-parallelism*. Typically, one of workers will be selected as the coordinator, so as to update parameters and then synchronize all copies at the global barrier.

Uniform APIs. We illustrate the programming APIs used in *Flegel* in Fig. 5, where users can override them to meet their own requirements. The function of *initParameter*() describes about how to give an initial guess for parameter $\theta^{(0)}$, as the input of the 1st iteration. *updateDataPoint*() is responsible for computing a given data point based on parameters refined in the previous iteration, and then returning an aggregator containing the change of local statistics, which can be immediately accumulated through invoking *aggregate*(). *aggregate*() will be called again at the flexible barrier to aggregate reports from all workers to form global statistics. After that, we update parameters by invoking *updateParameter*() specified by users. The *terminationCheck*() function is invoked by the coordinator. It tells *Flegel* about whether or not to terminate iterations. Another *voteToHalt*() allows *Flegel* to exclude a data point in the next iteration. It is invoked in *updateDataPoint*() when our new decentralized convergence criterion is used. Here we leave out details of the last two functions and give them in Section 5.3.

5.2 Semi-Centralized Coordination

Note that stragglers proceed slowly not only when training data, but also when responding to barrier signals. However, in the early version of *Flegel*, FSP does not touch any details of the flexible barrier as a whole, and hence the computing capacity of fast workers is still underutilized. More specifically, it implements the barrier by a two-phase coordination protocol [28]. As shown in Fig. 6a, the coordinator first broadcasts signals to workers to pause local update. The latter respond by committing training results to the former. Once all reports are received, the coordinator again broadcasts signals, together with newly refined parameters and the interval if necessary, to wake workers from waiting sleep. Extra waiting costs might be incurred at the first phase, for fast workers like *Worker_1*. The reason is the centralized coordination policy where the coordinator asks all workers to complete the same workloads from the same time instance but ignores the different responding speeds, resembling pre-assigned workloads in BSP.

We improve resource utilization by a semi-centralized policy where the first phase is issued in a decentralized manner. Based on the uniformly received interval η, each worker launches an independent local thread to tell itself to pause update. As demonstrated in Fig. 6b, that allows the i-th worker to personalize its real barrier interval as $\eta \pm \delta_i$, by continuously monitoring the elapsed time of respond and the change of training speed (totally measured as δ_i). A straggler can execute a short interval, so as to commit results in advance and then the coordinator can receive reports at roughly the same time instance. The second phase is still issued by the coordinator in a centralized manner, to ensure workers to see the same parameters and interval if changed. Such a semi-centralized policy can transfer extra wait to extra update, which narrows the latency gap of received reports across workers.

5.3 Convergence Criterion

Flegel provides three default convergence criterions. They work based on different statistics collected from workers. The first is to limit the maximum number of iterations, which is simple but ignores the training quality. By contrast, a choice is to set an idle objective value. Although pre-setting such a target is difficult, it can guarantee a fair comparison among different parallel optimizations, since the benchmark ML algorithm always converges to the same point. Another in-practice alternative is to compare a given

error bound with the value difference between two consecutive iterations. But an important problem here is that the barrier interval in FSP is unknown and dynamically changeable. An extremely short interval, of course, leads to a small variation and hence a false-positive check result. We mitigate this problem by decoupling termination check and iteration control. The check interval is set to the runtime of performing a full pass computation over data on the slowest worker.

5.4 Boundary of Flegel

We finally discuss the boundary of our FSP design, to show what ML algorithms can be run on *Flegel*. FSP especially works well for the widely employed *data-parallelism* training policy where sample data points distributed across workers are computed to contribute to updating parameters. In most cases, data points as individual entities follow the Independently Identical Distribution (IID) assumption. FSP thereby can flexibly tune how many of them are consumed within an iteration to remove waiting costs. Currently, *Flegel* implements *data-parallelism* in the widely-used *parameter-server* manner. However, it can also be extended to support another *Ring-AllReduce* manner [38] to resolve network congestion, especially when training large pre-trained models with hundreds of billions parameters.

By contrast, there usually exist strict consistent constraints when updating model parameters as a whole in *model-parallelism* and scheduling stages in *pipeline-parallelism*, both of which impair the flexibility of FSP. Another exception is the scenario where IID does not hold true because of the application-related semantic-constraint. For example, action recognition in videos essentially requires consecutive frames associated with some action to be processed in the same batch, to preserve the inherent semantic coherence [39]. FSP might break this principle because a batch/iteration can be terminated at any time, which generates negative impacts on the model accuracy.

6 Performance Studies

We finally conduct extensive experiments to explore performance features of our proposals. All tests are implemented on our Java-based prototype system *Flegel*, to make an end-to-end comparison.

6.1 Experimental Setup

The general experimental setting details are given as below:

Frameworks for Comparison. We mainly report the performance of our basic binary search based FSP with the empirical initial interval like Ref. [24] (FSPB), and the smart multi-stage variant (FSPV) without sensitive initial input. We compare them against the basic full-batch BSP framework (BSPB) and its mini-batch variant (BSPV). The latter automatically seeks an optimal batch size based on cost-benefit analysis [10] at the very beginning of computations. Asynchronous Parallel frameworks (ASP) are also tested to investigate the importance of barriers on convergence.

Experimental Cluster. Our cluster consists of 8 slave machines with one additional master connected by a Gigabit Ethernet switch. Each slave is configured with 4 cores (Intel Xeon E-2224, 3.5 GHz) and 32 GB of RAM; while the master has 10 cores (Intel Core I9-10900 K, 3.7 GHz) and 64 GB of RAM. By default, a given ML job is divided into 16 workers evenly scheduled across 4 slaves.

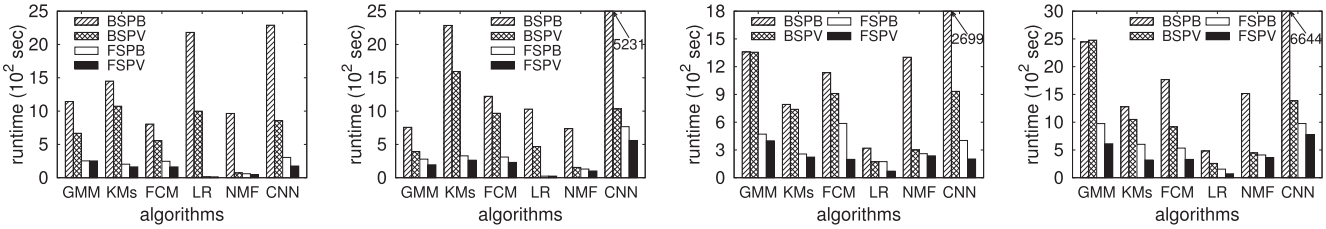
ML Algorithms and Datasets. Six representative algorithms are tested to explore performance features: K-Means (KMs), Fuzzy CMeans (FCM), and Gaussian Mixture Model (GMM), trained by EM; LR, Non-negative Matrix Factorization (NMF), and Convolution Neural Network (CNN),[3] trained by GD. These tests are run over publicly available datasets and synthetic data generated in a random manner. Table 1 shows the detailed combination cases. We group these cases into two categories by the size $\mathbb{P}\times\mathbb{D}$, each of which is an ordered 6-tuples w.r.t. the algorithm list. The Larger Dataset Tuple *LDT* is {PUF, HIGGS, HIGGS, DET, ML25 M, MNIST}; while the smaller one *SDT* is {SUSY, MASS, MASS, HIGGS, ML10 M, SYN}. Note that NMF and CNN take as input a large matrix and a set of images, respectively. Accordingly, $\mathbb{P}$ indicates the number of rows/images, while $\mathbb{D}$ indicates the number of columns/pixels.

Evaluation Metrics. Two metrics are evaluated in experiments, *runtime* and *objective*function value. The former is defined as the elapsed time of iterative training. Loading data and dumping results are excluded as they are the same for all frameworks. Given an algorithm, different frameworks enjoy the same settings, like initial centroid and learning rate, for a fair comparison. Besides, we employ the second criterion in Section 5.3, i.e., pre-setting the target of *objective*, to ensure that a ML algorithm under different frameworks can always converge to the same point.

Experiment Design. As stated in Ref. [14], it is hard to instrument all real distributed infrastructures to accurately distinguish massive and very different straggling factors, although they indeed happen in practice. We thereby experiment with Injected and Naturally-occurring straggler patterns (*Injected* and *Natural*). From literature researches [13], [14], [17], [40], *Injected* is to suspend training threads on given workers for the τ milliseconds every 1,000 data points processed. Through varying τ, we can control the straggling skewness from general-encountered patterns to extreme-stress test patterns. *Natural* is the skewed distribution of workers among slaves in the multi-tenant scenario [13], [41]. This happens because a job (and its related workers) might terminate at any time once it converges, which dynamically generates the skewness for remaining running jobs. The specific simulation is the same as described in Section 1. Clearly, workers with inserted suspending or on congested slaves, immediately become stragglers.

It is worth noting that *Injected* and *Natural* respectively stand for the periodic transient and the static persistent stragglers, since the former periodically/dynamically happens while the latter generates resource contention in the whole training process. They can validate the generality of our solutions. Besides, when exploring the reason of runtime gains, we usually give some representative results and omit others for brevity and the limited manuscript space, since they exhibit the similar phenomenons.

3. It has two convolutional layers with 6 and 12 5×5 kernels respectively.



(a) SDT & Injected (b) LDT & Injected (c) SDT & Natural (d) LDT & Natural

Fig. 7. Runtime evaluation with stragglers (BSP versus FSP).

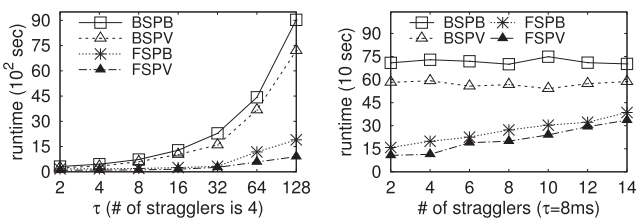
6.2 Effectiveness of FSP

We first test all of combination cases in two scenarios: one is the heterogeneous environment (*Injected*&*Natural*) where FSPB generally wins BSPV as demonstrated in Fig. 7, as the former's flexible mechanism removes expensive waiting costs; the other one is the homogeneous cluster status where BSPV and FSPB have comparable performance as shown in Fig. 8, because they both can smartly compute a proper batch size or barrier interval at the very beginning of iterations. Benefitting from flexibility and multi-stage interval adjustor, FSPV is always the best. The reasons why it beats FSPB are twofold: (1) the interval that yields the best speedup during initial iterations might not be the optimal setting in the whole iterations but multi-stage adjusting can overcome this shortcoming; (2) its new adjustor is initial-input-interval-free and can quickly output a setting by one-pass analysis, which effectively avoids false positive comparison in FSPB's recursive binary adjustor. By contrast, BSPB consistently works the worst, due to the parameter update delay and the expensive waiting costs.

In particular, the speedup of FSPV compared with BSPB is up to the factors of 12X (from 1.5X) and 170X (from 3.4X) in the two distinguished scenarios. Even compared with BSPV and FSPB, FSPV can still respectively offer 1.3-78X and 1.0-3.0X speedups when stragglers happen. The homogeneous setting w/o stragglers weakens the competitive advantage of flexible barrier, and hence FSPV only runs 1.3X faster on average than competitors.

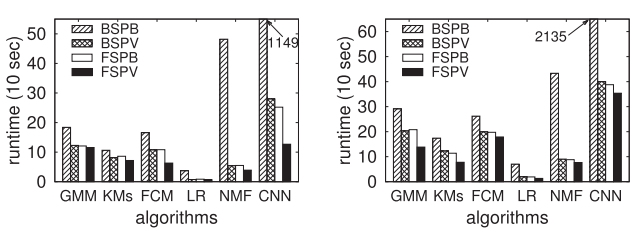
We next investigate the scalability when varying the degree of skewness and the number of stragglers, so that we can give a preferred setting in *Injected*. As an indication of how different frameworks scale with τ, Fig. 9a depicts the runtime variation. BSPB and BSPV essentially get performance degradation because of the increasing waiting costs—dominated by the slowest workers. While, FSPB and FSPV scale well, that mainly stems from the flexible barrier design. Fig. 9b plots runtime against the number of stragglers. The performance gap between BSP- and FSP-based frameworks clearly narrows down. This is because the former is insensitive to how many workers are slow, while for the latter, more data are traversed with fewer passes, which forces ML algorithms to make a long detour to converge. Finally, our general comparison finds that *Injected* with $\tau = 32$ms and 4 stragglers can offer a comparable degradation to *Natural*, for all tested frameworks. We thereby use this combination as the default setting through the paper.

We now explore the resource utilization of the CPU power, to emphasize the advantage of our FSP design. We analyze utilization of different frameworks by computing their synchronization costs, since CPUs within workers pause only at the synchronization barriers. We define the cost as the elapsed time from the point where a worker enters a barrier at the end of an iteration, to the point where it leaves the barrier. Smaller elapsed time clearly indicates higher utilization. Fig. 10a first demonstrates the time difference between the fastest worker and the slowest one, for each test case. The significant difference in values from BSP-based frameworks indicates that fast workers must wait for stragglers for a long time, which heavily impairs the utilization of CPUs. Fig. 10b further depicts the detailed skewed elapsed time distribution for every worker. In BSPV, fast workers (with ids from 5 to 16) arrive at the barrier point in advance, but are blocked until stragglers (with ids from 1 to 4) have completed pre-assigned workloads. The former thereby have large synchronization costs, wasting a lot of



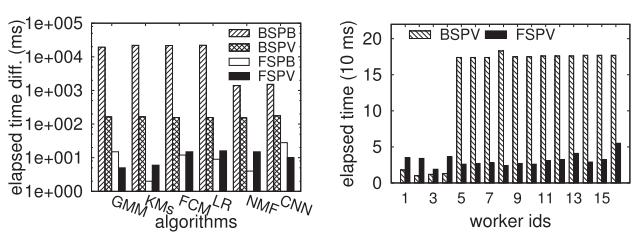
(a) Interval (b) Stragglers

Fig. 9. Impact of configurations in *Injected* (KMs over HIGGS).



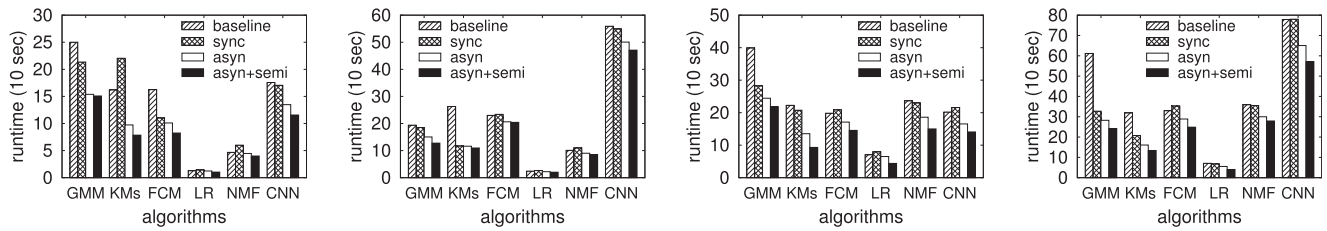
(a) SDT (b) LDT

Fig. 8. Runtime evaluation without stragglers.



(a) Overall comparison of sync. costs (b) Distribution of sync. costs

Fig. 10. Utilization of the CPU power (*SDT* & *Injected* with straggler ids from 1 to 4; sub-figure(b) is tested on KMs over HIGGS).



(a) SDT & Injected (b) LDT & Injected (c) SDT & Natural (d) LDT & Natural

Fig. 11. Runtime evaluation with stragglers (FSPV as baseline: sync. versus asyn. data migration, and centralized versus semi-centralized coordination).

CPU resources. By contrast, FSPV enables both fast and slow workers to enter the barrier at nearly the same time, to reduce the skewness and hence improve CPU utilization.

6.3 Effectiveness of Other Optimizations

We then verify the effectiveness of our lightweight asynchronous workload balance and semi-centralized coordination techniques, termed as *asyn* and *semi* for short. For the former, we test the synchronous implementation as a counterpart, i.e., *sync*, to show the impact of data migration costs. Since FSPV is the best solution as discussed in Section 6.2, here we use it as *baseline*.

As presented in Fig. 11, *asyn* can largely improve the performance of *baseline* by up to 55.8% (KMs on HIGGS, *Injected*). It also consistently outperforms *sync*, because the benefit of the latter is usually offset by its expensive blocking migration cost, which even generates 35.9% performance degradation in the extreme case (KMs on HIGGS, *Injected*). The non-deterministic impact prohibitively prevents us from safety using it as a default optimization choice. *semi* creates another performance gap between *baseline* and *asyn*. Compared with *asyn*, the best runtime improvement is 32.5% at most (LR on HIGGS, *Natural*). Overall, together with the two optimizations, the speedup of FSPV versus the up-to-date FSPB is increased to 4.5X (from 1.2X).

For better understanding how our techniques work, we next give detailed behavior analysis.

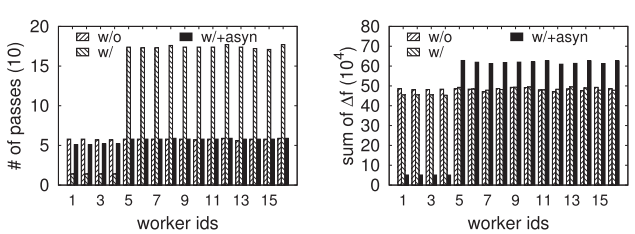
Asynchronous Data Migration. After a ML algorithm converges, we report two metrics for each of total 16 workers: the number of completed full traversal passes, indicating training speed; and the accumulated changing contributions w.r.t. *objective*, indicating training quality. We use two scenarios w/o and w/ injected stragglers to demonstrate the metric variation. As shown in Fig. 12, although workers proceed at very different speeds, the summation of Δf just slightly skews in the favor of fast workers. This is not surprising because data on stragglers are always updated based on up-to-date parameters, which generates a large quality per update and hence the summation, even with the reduced update number. However, by balancing the speed, the sufficient power of fast workers can be used to train high-quality data originally residing on stragglers. These data make full contributions to convergence, which yields an overall success of increasing the total summation of Δf from all workers.

Semi-Centralized Coordination. Its performance gain is actually complex, because the reduced barrier cost can also enable more frequent parameter update (smaller η) and then accelerate the whole training. Here we distinguish the two impacts by recording the synchronization cost per iteration, respectively with traditional centralized and our semi-variant policies. Reports in Fig. 13 reveal that our proposal generally beats the counterpart. For example, the total runtime reduction for KMs on HIGGS is 26.8secs, where the barrier optimization contributes 56.6%, i.e., 15.2secs, and the other gain is due to fresh parameters. Fig. 13b depicts another interesting fact. We find that although the reduced η increases the number of iterations, our efficient coordination policy still makes the accumulated synchronization cost decrease from 48.4secs down to 38.3secs. The remarkable performance contribution (23.9%) in the total runtime reduction (42.3secs) again validates that our design is cost-effective.

6.4 Empirical Validation of Convergence

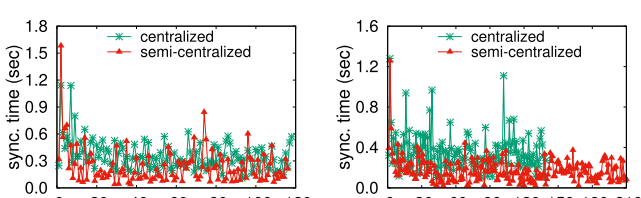
Besides a formal convergence proof in Section 3.2, now we give an empirical validation by showing the *objective* variation during iterations. All tests are performed without stragglers since the convergence guarantee does not care about such a setting.

Fig. 14 reports *objective* versus elapsed time plots, which is achieved by continuously monitoring the computation progress at sampled time instances. Clearly, the two sub-fig-



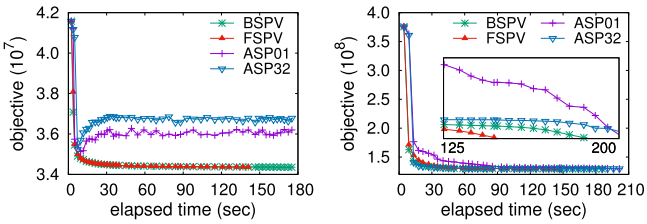
(a) Traversal passes (b) Contributions to convergence

Fig. 12. Effectiveness of asyn. data migration w/ and w/o stragglers (KMs on HIGGS, *Injected* with $\tau = 32$ms and straggler ids from 1 to 4).



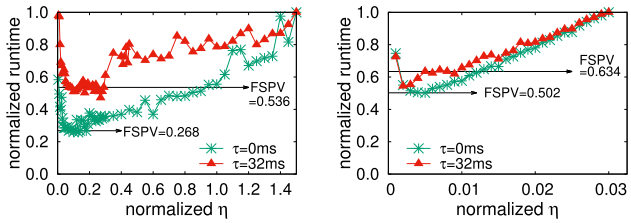
(a) KMs on HIGGS (b) KMs on MASS

Fig. 13. Effectiveness of semi-centralized coordination (*Natural*).



(a) KMs on HIGGS (b) FCM on MASS

Fig. 14. Convergence evaluation without stragglers.



(a) KMs on HIGGS (b) NMF on ML25M

Fig. 15. Effectiveness of multi-stage self-corrected adjustor (*Injected*).

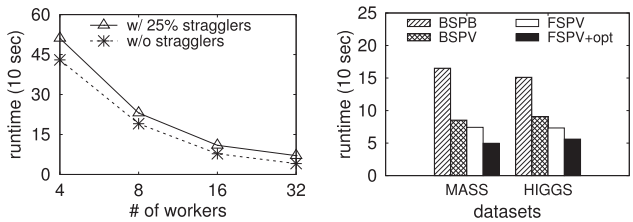
ures reveal that our FSPV monotonously decreases *objective*, like the traditional BSPV. Hence, algorithms under *Flegel* can converge to the correct solution, but more importantly, they exhibit a faster convergence speed than existing BSP-based implementations.

We also implement asynchronous frameworks (ASP) by completely removing global barriers [42]. In ASP, we manually set two numbers of parameter update batches (01 and 32) for data on each worker, to explore its performance features. There is no doubt that ASP largely mitigates synchronization costs, but this benefit usually cannot translate to fast convergence. This is because asynchronous updates on global parameters make some workers use stale local copies. The incurred errors make it extremely difficult to theoretically analyze the convergence, and indeed ASP-based algorithms can readily diverge (like KMs in Fig. 14a). Thus, ASP is not a preferred solution even though it works well in some cases. In contrast, our proposals provide enough flexibility for ML algorithms to be highly efficient.

Another observation we find is that for KMs under ASP, a small batch number (i.e., a large batch size) works well. However, this does not hold true for FCM. Selecting a proper batch size in ASP is also a big challenge for end-users. We cannot easily design an adjustor as used in FSPV to uniformly deal with the thoroughly opposite phenomenons.

6.5 Empirical Validation of Optimal Intervals

Previous experiments have confirmed that FSPV outperforms FSPB, because of the multi-stage interval adjustor. Now we investigate its effectiveness in detail, i.e., how likely can it seek η that is optimal or close to optimal. To find a real optimal interval, we repeatedly run ML algorithms and manually set a possible barrier interval every time. Fig. 15 reports the runtime normalized w.r.t. the maximum in all attempts, against the selected interval normalized w.r.t. the runtime of a full-batch iteration. In particular, we run tests under *Injected* with two τ values, so as to simulate different compute cluster statuses.



(a) CPU (HIGGS) (b) CPU+GPU

Fig. 16. Scalability with different compute resources (KMs, *Injected*).

With η being increased, we can easily observe that the runtime first quickly decreases and then gradually increases in all cases. This can be explained by the tradeoff between high synchronization costs incurred by frequent barriers (small interval) and low training quality due to infrequent barriers (large interval). Note that the runtime curve varies with the specific combination of ML algorithms, datasets and the computing cluster status. It is impossible to get an one-fit-all solution to find the "sweet spots," by offline analysis, as used in Ref. [10]. This figure also tells us that simply setting an interval (like 0.5 in Ref. [41]) might generate significant performance loss. By contrast, FSPV shows prominent robustness. It can always find a better compromise by online adjusting intervals and correcting prediction errors. The average runtime reduction is up to 50% as reported. When normalized by BSPV w/ the two τ values, the runtime of FSPV varies from 0.633 to 0.250 for KMs, and 0.849 to 0.226 for NMF. That again shows the remarkably outstanding performance of our proposals, especially when stragglers happen.

6.6 Scalability Evaluation

Let FSPV+*opt* stand for FSPV with the asynchronous data migration and semi-centralized coordination optimizations. We finally investigate its scalability using different hardware configurations. Fig. 16 explores the impacts of compute resources. In sub-figure(a), another 4 slaves with the same configuration are added into our cluster. We increase the number of workers from 4 to 32 and always evenly schedule them across the total 8 slaves. Take KMs on HIGGS as an example. FSPV+*opt* achieves a nearly linear speedup, regardless of the presence of stragglers. On the other hand, sub-figure(b) shows the runtime performance with 5 workers, one of which is run on the NVIDIA Geforce RTX 3080 GPU (8704 CUDA cores, 1.44 GHz, and 10 GB GDDR6X memory) and others are run with CPUs. The GPU device with massive threads has strong computing power, yielding an inherent heterogeneity gap compared with CPU. FSPV beats BSPV because GPU can train local data more frequently, instead of blocking itself to wait for CPUs. Together with the optimizations of data migration and semi-synchronization, FSPV+*opt* finally improves the overall performance by 42% at most.

We also test the performance of our proposals under different network bandwidths. By manually limiting the upper bound of network throughput, we create two test scenarios: Non-limited and Limited. The bandwidth of the latter decreases by roughly 50%, as reported by the benchmarking tool *iPerf*. For FSPV, we find that the speedup degradation is less than 2%, which can be ignored. This is because now ML

TABLE 2
Scalability of the Asynchronous Data Migration With Different Network Bandwidths (KMs on HIGGS, MB/s)

Settings	Non-limited			Limited		
	iPerf	*Only*	*Normal*	*iPerf*	*Only*	*Normal*
Injected	116.75	99.72	98.80	59.38	57.57	56.89
Natural	108.13	71.51	69.22	59.69	55.53	54.74

algorithms used in our testbed experiments have the limited size of model parameters. Collecting statistics and broadcasting parameters generate tiny runtime delay, which is not very sensitive to the bandwidth change. However, the performance penalty for the asynchronous data migration speed is significant, since many data points are migrated across workers. Note that CPU is essentially involved in transferring data. In order to distinguish the impact of CPU resource contention, we submit an empty job as a baseline, which does nothing but *Only* transfers equal-size data as done in a *Normal* job. Table 2 summarizes the behaviors by reporting the peer-to-peer network throughput and data migration speed. The migration speed clearly linearly decreases with the bandwidth reduction. However, the overall performance degradation is still negligible (less than 4%), because our bi-directional sliding window and on-demand interruption techniques in Section 4.3 can avoid the loss of traversing passes even though data are flying on network for a long while. In particular, *Normal* always achieves comparable performance to *Only*. That explicitly validates that our asynchronous design can effectively overlap the communication-intensive data delivery and the compute-intensive local training.

7 Related Works

Today's distributed systems enable scalable ML computations in the Big Data era. There are a flurry of efforts targeted at solving the straggler problem for further performance enhancement, especially in heterogeneous environments. Below, we review these works from three perspectives to highlight our contributions.

Data Migration and Replication. A straightforward straggler solution is that within each iteration, idle workers that have already reached the pre-defined barrier location, dynamically steal workloads from busy workers [12], [13]. In particular, Wang et al. also aim to balance the traversal number by continuously exchanging data between the fastest and the slowest workers [41]. However, all of them migrate data at runtime in a blocking manner, which is expensive as reported in our experiments. Harlap et al. [14] propose to group workers and pre-replicate data on each other before iterations, so as to quickly migrate logic computation tasks, rather than data. In addition, Hadoop and Spark [43], [44] support heavyweight speculative execution by training data on straggling workers redundantly and using the output from the first successful run. Tandon et al. collect encoded gradients for accurate parameter update from partial fast workers with redundant replications and computations [45]. Clearly, these three policies require additional memory or compute resources.

Relaxed Synchronous Constraint. Synchronous ML implementations can provide strict convergence guarantee, but suffer from stragglers due to the global pre-defined barriers. Early researchers tackle this problem by relaxing the synchronous constraint. Their works basically fall into three categories. (1) The first one is confined synchronous parallel (CSP). It confines the barrier operation to a subset of workers, which naturally reduces the number of workers blocked by stragglers. Chen et al. [15] directly skip the top-k slowest workers and drop their training results. Zheng et al. allow slow workers to continue training with stale parameters and commit results in future [17]. Miao et al. give a formal analysis in the All-Reduce parallel setting with point-to-point communication pattern [46]. Ref. [16] and Ref. [47] both focus on dynamically tune k, based on runtime statistics. (2) Another extreme is asynchronous parallel (ASP) without any barrier [21], [22], [23]. Each worker works independently and hence usually uses different versions of parameters. (3) The third branch makes a compromise by stale synchronous parallel (SSP) [18]. It is similar to ASP but the iteration number gap between the fastest worker and the slowest one cannot exceed a given threshold. Further, Shi et al. propose to replace the worker that is always detected as a straggler [19]; Zhou et al. design coarse-grained SSP by grouping similar workers into communities [20]; Jiang et al. achieve another improvement by tuning learning rate [48]; and Atallah et al. give a hybrid framework based on coding gradients [45] and SSP [49].

The relaxing idea works well for transient stragglers, but cannot easily cope with persistent stragglers. CSP and ASP reduce contributions from data on such workers, and hence, the model might overfit and/or converge slowly; while, SSP inevitably degrades to BSP after the threshold is exceeded. Besides, when training data with inconsistent parameters, ML requires more iterations to tolerate incurred errors, and even diverges.

Tunable Workload Assignment. Tuning workloads per iteration can also significantly affect overall training performance. The most important study is to transform full-batch update into mini-batch variant [1], [4], to use newly refined parameters in remaining training, as quickly as possible. In particular, our EM convergence proof is based on the work of Neal et al. [1], but goes further. We prove that in distributed environments, the convergence can be guaranteed by sharing consistent parameters across workers. Pre-establishing barrier locations is thereby unnecessary. Further, many efforts have been devoted into identifying the mini-batch size [2], [3], [10], all of which ignore the impact of stragglers. Recent studies mitigate this problem by customizing batch size for different workers [25], [26], [27], [50], instead of using the uniform setting. However, the size is still measured by the number of data points, and is pre-defined based on analyzing historical information yet prior to the upcoming iteration. That is very different from our purely time-based design, and cannot immediately react to transient stragglers, especially for those suddenly happening in the current iteration. They also notice the skewed distribution of traversal numbers, and embrace the impact by tuning learning rate and/or gradient weight [25], [26], [46] in GD, which should be empirically and carefully adjusted by multiple times. And the algorithm-specific setting

cannot be easily generalized. Resembling our *Flegel*, some works also employ a coordinator to synchronize workers [13], [24], [41]. The synchronizing location does not care about the specific progress of each worker, but, instead, is dominated by the total number of data already processed. An accurate barrier control requires frequent and fine-grained statistic collection from all workers, which wastes network bandwidth. Worse, their counting-based interval is either simply set as a half of total data [41] or recursively adjusted like our binary searching [24], both of which can not work well as shown in our experiments.

Other Complementary Optimizations. We also investigate many important but complementary works, like prioritized training [51], flexible parallelism tuning in task-level [52] and thread-level within a task [53], and elastic resource scheduling [54], [55]. Besides, Chen et al. attempt to solve the problem of parameter server stragglers by dynamically scaling in/out servers and re-balancing the parameter distribution [56], which is beyond the scope of our focus about worker stragglers. Further, Wang et al. conduct local training and parameter propagation in two separate threads for each worker, so as to overlap computation and communication [57]. Overall, all of them can be plugged into our *Flegel* system. Note that Ref. [53] also proposes asynchronous data migration, but it targets at balancing the runtime distribution, instead of the traversing frequency. The latter is a key factor dominating the performance of FSP, as evidenced by our experiments.

8 Conclusion and Future Works

This paper investigates the efficiency of distributed ML computations. We propose a new flexible synchronous parallel (FSP) framework that enables fast workers to perform more useful works, instead of blocking themselves to wait for stragglers. The built-in adjustors can automatically seek a proper synchronous interval to maximize the efficiency gain. Our lightweight dynamic workload balance technique further boosts the performance by equalizing the data traversal passes.

For several heterogeneous configurations that we tested, FSP consistently performs the best. However, the testing space needs further exploration, like the flexible coordination across multiple GPUs and even the fine-grained warps/blocks in a single GPU. On the other hand, although we implement FSP in our own prototype system *Flegel*, as discussed in Section 5.4, any system that supports *data-parallelism* can benefit from our design. Take popular open-source ML frameworks TensorFlow and PyTorch as examples. We can embed FSP in them by respectively rewriting *tf.train.SyncReplicasOptimizer* and *torch.nn.parallel.DistributedDataParallel&torch.distributed*, to validate its effectiveness in open-source communities. Overall, we believe that FSP is an interesting idea for fast ML computation. We plan to investigate these open questions as future works.

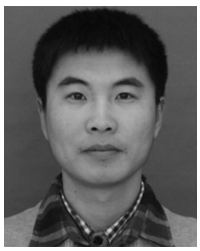
Acknowledgments

Zhigang Wang and Yilei Tu are co-first authors.

References

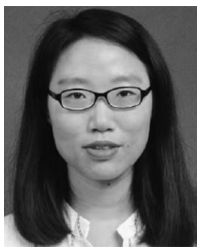
[1] R. M. Neal and G. E. Hinton, "A view of the EM algorithm that justifies incremental, sparse, and other variants," in *Learning in Graphical Models*. Berlin, Germany: Springer, 1998, pp. 355–368.
[2] H. Daneshmand, A. Lucchi, and T. Hofmann, "Starting small-learning with adaptive sample sizes," in *Proc. 33rd Int. Conf. Mach. Learn.*, 2016, pp. 1463–1471.
[3] S. De, A. K. Yadav, D. W. Jacobs, and T. Goldstein, "Automated inference with adaptive batches," in *Proc. 20th Int. Conf. Artif. Intell. Statist.*, 2017, pp. 1504–1513.
[4] M. Li, T. Zhang, Y. Chen, and A. J. Smola, "Efficient mini-batch training for stochastic optimization," in *Proc. 20th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2014, pp. 661–670.
[5] J. Wolfe, A. Haghighi, and D. Klein, "Fully distributed EM for very large datasets," in *Proc. 25th Int. Conf. Mach. Learn.*, 2008, pp. 1184–1191.
[6] A. J. Smola and S. M. Narayanamurthy, "An architecture for parallel topic models," *Proc. VLDB Endowment*, vol. 3, no. 1, pp. 703–710, 2010.
[7] E. R. Sparks et al., "MLI: An API for distributed machine learning," in *Proc. IEEE 13th Int. Conf. Data Mining*, 2013, pp. 1187–1192.
[8] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.
[9] S. Subramanya, T. Guo, P. Sharma, D. Irwin, and P. Shenoy, "SpotOn: A batch computing service for the spot market," in *Proc. 6th ACM Symp. Cloud Comput.*, 2015, pp. 329–341.
[10] J. Yin, Y. Zhang, and L. Gao, "Accelerating expectation-maximization algorithms with frequent updates," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2012, pp. 275–283.
[11] A. Wang et al., "FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at Alibaba cloud function compute," in *Proc. USENIX Annu. Tech. Conf.*, 2021, pp. 443–457.
[12] U. Acar, A. Charguéraud, and M. Rainey, "Scheduling parallel programs by work stealing with private deques," in *Proc. ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2013, pp. 219–228.
[13] S. Li, J. Xue, Z. Yang, and Y. Dai, "Efficient distributed machine learning with trigger driven parallel training," in *Proc. IEEE Glob. Commun. Conf.*, 2016, pp. 1–6.
[14] A. Harlap et al., "Addressing the straggler problem for iterative convergent parallel ML," in *Proc. 7th ACM Symp. Cloud Comput.*, 2016, pp. 98–111.
[15] X. Pan, J. Chen, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous SGD," 2017, *arXiv:1702.05800*.
[16] C. Xu, G. Neglia, and N. Sebastianelli, "Dynamic backup workers for parallel machine learning," *Comput. Netw.*, vol. 188, 2021, Art. no. 107846.
[17] M. Zheng, D. Mao, L. Yang, Y. Wei, and Z. Hu, "DOSP: An optimal synchronization of parameter server for distributed machine learning," *J. Supercomput.*, vol. 78, pp. 13865–13892, 2022.
[18] Q. Ho et al., "More effective distributed ML via a stale synchronous parallel parameter server," in *Proc. 27th Annu. Conf. Neural Inf. Process. Syst.*, 2013, pp. 1223–1231.
[19] H. Shi, Y. Zhao, B. Zhang, K. Yoshigoe, and A. V. Vasilakos, "A free stale synchronous parallel strategy for distributed machine learning," in *Proc. Int. Conf. Big Data Eng.*, 2019, pp. 23–29.
[20] Q. Zhou et al., "Petrel: Heterogeneity-aware distributed deep learning via hybrid synchronization," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 5, pp. 1030–1043, May 2021.
[21] B. Recht, C. Re, S. Wright, and F. Niu, "HOGWILD: A lock-free approach to parallelizing stochastic gradient descent," in *Proc. 25th Annu. Conf. Neural Inf. Process. Syst.*, 2011, pp. 693–701.
[22] M. Li et al., "Scaling distributed machine learning with the parameter server," in *Proc. 11th USENIX Symp. Oper. Syst. Des. Implementation*, 2014, pp. 583–598.
[23] S. Soori, B. Can, M. Gürbüzbalaban, and M. M. Dehnavi, "ASYNC: A cloud engine with asynchrony and history for distributed machine learning," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2020, pp. 429–439.
[24] G. Zhao, T. Zhou, and L. Gao, "A proactive data-parallel framework for machine learning," in *Proc. IEEE/ACM 8th Int. Conf. Big Data Comput.*, 2021, pp. 69–79.
[25] C. Chen, W. Wang, and B. Li, "Round-robin synchronization: Mitigating communication bottlenecks in parameter servers," in *Proc. IEEE Conf. Comput. Commun.*, 2019, pp. 532–540.

[26] C. Chen, Q. Weng, W. Wang, B. Li, and B. Li, "Accelerating distributed learning in non-dedicated environments," *IEEE Trans. Cloud Comput.*, to be published, doi: 10.1109/TCC.2021.3102593.
[27] X. Zhao, M. Papagelis, A. An, B. X. Chen, J. Liu, and Y. Hu, "ZipLine: An optimized algorithm for the elastic bulk synchronous parallel model," *Mach. Learn.*, vol. 110, no. 10, pp. 2867–2903, 2021.
[28] Z. Wang, L. Gao, Y. Gu, Y. Bao, and G. Yu, "FSP: Towards flexible synchronous parallel framework for expectation-maximization based algorithms on cloud," in *Proc. 8th ACM Symp. Cloud Comput.*, 2017, pp. 1–14.
[29] T. Zhang, "Solving large scale linear prediction problems using stochastic gradient descent algorithms," in *Proc. 21st Int. Conf. Mach. Learn.*, 2004, pp. 9–16.
[30] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the EM algorithm," *J. Roy. Statist. Soc. Ser. B Methodol.*, vol. 39, pp. 1–22, 1977.
[31] J. C. Duchi and Y. Singer, "Efficient learning using forward-backward splitting," in *Proc. 23rd Annu. Conf. Neural Inf. Process. Syst.*, 2009, pp. 495–503.
[32] R. T. McDonald, K. B. Hall, and G. Mann, "Distributed training strategies for the structured perceptron," in *Proc. Hum. Lang. Technol.: Conf. North Amer. Chapter Assoc. Comput. Linguistics*, 2010, pp. 456–464.
[33] K. Chen and Q. Huo, "Scalable training of deep learning machines by incremental block training with intra-block parallel optimization and blockwise model-update filtering," in *Proc. IEEE Int. Conf. Acoust. Speech Signal*, 2016, pp. 5880–5884.
[34] S. P. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Found. Trends Mach. Learn.*, vol. 3, no. 1, pp. 1–122, 2011.
[35] P. Chang et al., "An unprecedented set of high-resolution earth system simulations for understanding multiscale interactions in climate variability and change," *J. Adv. Model. Earth Syst.*, vol. 12, no. 12, pp. 1–52, 2020.
[36] L. Bottou, F. E. Curtis, and J. Nocedal, "Optimization methods for large-scale machine learning," *Soc. Ind. Appl. Math. Rev.*, vol. 60, no. 2, pp. 223–311, 2018.
[37] B. Thiesson, C. Meek, and D. Heckerman, "Accelerating em for large databases," *Mach. Learn.*, vol. 45, no. 3, pp. 279–299, 2001.
[38] A. Sergeev and M. D. Balso, "Horovod: Fast and easy distributed deep learning in tensorflow," 2018, *arXiv:1802.05799*.
[39] L. Wang et al., "Temporal segment networks: Towards good practices for deep action recognition," in *Proc. 14th Eur. Conf. Comput. Vis.*, 2016, pp. 20–36.
[40] D. Yang, W. Rang, and D. Cheng, "Mitigating stragglers in the decentralized training on heterogeneous clusters," in *Proc. 21st Int. Middleware Conf.*, 2020, pp. 386–399.
[41] S. Wang, W. Chen, A. Pi, and X. Zhou, "Aggressive synchronization with partial processing for iterative ML jobs on clusters," in *Proc. 19th Int. Middleware Conf.*, 2018, pp. 253–265.
[42] T. M. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project adam: Building an efficient and scalable deep learning training system," in *Proc. 11th USENIX Symp. Oper. Syst. Des. Implementation*, 2014, pp. 571–582.
[43] Apache hadoop, 2022. [Online]. Available: http://hadoop.apache.org/
[44] Apache spark, 2022. [Online]. Available: http://spark.apache.org/
[45] R. Tandon, Q. Lei, A. G. Dimakis, and N. Karampatziakis, "Gradient coding: Avoiding stragglers in distributed learning," in *Proc. 34th Int. Conf. Mach. Learn.*, 2017, pp. 3368–3376.
[46] X. Miao et al., "Heterogeneity-aware distributed machine learning training via partial reduce," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2021, pp. 2262–2270.
[47] H. Yu, Z. Zhu, X. Chen, Y. Cheng, Y. Hu, and X. Li, "Accelerating distributed training in heterogeneous clusters via a straggler-aware parameter server," in *Proc. IEEE 21st Int. Conf. High Perform. Comput. Commun.*, 2019, pp. 200–207.
[48] J. Jiang, B. Cui, C. Zhang, and L. Yu, "Heterogeneity-aware distributed parameter servers," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2017, pp. 463–478.
[49] E. Atallah, N. Rahnavard, and C. Enyioha, "Straggler-robust distributed optimization in parameter-server networks," 2021, *arXiv:2007.13688*.
[50] R. Han, S. Li, X. Wang, C. H. Liu, G. Xin, and L. Y. Chen, "Accelerating gossip-based deep learning in heterogeneous edge computing platforms," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 7, pp. 1591–1602, Jul. 2021.
[51] T. Zhou and L. Gao, "Distributed framework for accelerating training of deep learning models through prioritization," in *Proc. IEEE Int. Conf. Cloud Eng.*, 2021, pp. 201–209.
[52] Y. Huang et al., "FlexPS: Flexible parallelism control in parameter server architecture," *Proc. VLDB Endowment*, vol. 11, no. 5, pp. 566–579, 2018.
[53] Q. Zhou et al., "Falcon: Addressing stragglers in heterogeneous parameter server via multiple parallelism," *IEEE Trans. Comput.*, vol. 70, no. 1, pp. 139–155, Jan. 2021.
[54] A. Qiao et al., "Litz: Elastic framework for high-performance distributed machine learning," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 631–644.
[55] S. Wang, A. Pi, and X. Zhou, "Elastic parameter server: Accelerating ML training with scalable resource scheduling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 5, pp. 1128–1143, May 2022.
[56] Y. Chen, Y. Peng, Y. Bao, C. Wu, Y. Zhu, and C. Guo, "Elastic parameter server load distribution in deep learning clusters," in *Proc. ACM Symp. Cloud Comput.*, 2020, pp. 507–521.
[57] H. Wang, S. Guo, and R. Li, "OSP: Overlapping computation and communication in parameter server for fast machine learning," in *Proc. 48th Int. Conf. Parallel Process.*, 2019, pp. 82:1–82:10.



Zhigang Wang received the PhD degree in computer software and theory from Northeastern University, China, in 2018. He is currently an associate professor with the School of Computer Science and Technology, Ocean University of China. His research interests include distributed graph processing and machine learning. He is a member of the China Computer Federation (CCF). He received the CCF Outstanding Doctoral Dissertation Award in 2018 and ACM QINGDAO Rising Star Award in 2019.

Yilei Tu received the bachelor's degree from the School of Computer and Information, Anhui Polytechnic University, China, in 2020. He is currently working toward the master's degree with the School of Computer Science and Technology, Ocean University of China. His research interests include Big Data processing and distributed machine learning.



Ning Wang received the PhD degree in computer software and theory from Northeastern University, China, in 2017. She is currently a lecture with the School of Computer Science and Technology, Ocean University of China. She has been a visiting PhD student with Nanyang Technological University during December 2014 to December 2015. Her research interests include Big Data process, differential privacy protection, and machine learning.

Lixin Gao (Fellow, IEEE) received the PhD degree in computer science from the University of Massachusetts at Amherst, in 1996. Now, she is a professor of Electrical and Computer Engineering at the University of Massachusetts at Amherst. Her research interests include social networks, Internet routing, network virtualization and cloud computing. Between May 1999 and January 2000, she was a visiting researcher with AT&T Research Labs and DIMACS. She was an Alfred P. Sloan fellow between 2003-2005 and received an NSF CAREER Award in 1999. She won the Best Paper Award from IEEE INFOCOM 2010 and ACM SoCC 2011, and the test-of-time Award in ACM SIGMETRICS 2010. She received the Chancellors Award for Outstanding Accomplishment in Research and Creative Activity in 2010, and is a fellow of the ACM.

Jie Nie received the PhD degree in engineering from the Ocean University of China, in 2011. In 2013, she entered the Computer Science and Technology Postdoctoral Research Station, Tsinghua University for postdoctoral research. In 2019, she worked with the Industrial Internet Research Institute as an associate professor, in Department of Information Science and Engineering, Ocean University of China. Her research interests include multimodal Big Data mining, multimedia content analysis, artificial intelligence, and marine environment forecasting.

Zhiqiang Wei (Member, IEEE) received the PhD degree from Tsinghua University, China, in 2001. He is currently a professor with the Ocean University of China. He is also the director of High Performance Computing Center, Pilot National Laboratory for Marine Science and Technology (Qingdao). His current research interests include the fields of intelligent information processing, social media, and Big Data analytics. He is a member of the CCF.

Yu Gu received the PhD degree in computer software and theory from Northeastern University, China, in 2010. Currently, he is a professor and the PhD supervisor with Northeastern University, China. His current research interests include Big Data analysis, spatial data management, and graph data management. He is a senior member of the China Computer Federation (CCF).

Ge Yu (Member, IEEE) received the PhD degree in computer science from the Kyushu University of Japan, in 1996. He is currently a professor and the PhD supervisor with Northeastern University of China. His research interests include distributed and parallel database, OLAP and data warehousing, data integration, graph data management, etc. He is a member of the IEEE Computer Society, ACM, and a fellow of the China Computer Federation (CCF).