FSP: Towards Flexible Synchronous Parallel Framework for Expectation-Maximization based Algorithms on Cloud

Zhigang Wang[1] Lixin Gao[2] Yu Gu[1] Yubin Bao[1] Ge Yu[1]
[1]Northeastern University, China
[2]University of Massachusetts Amherst, USA
wangzhiganglab@gmail.com, lgao@ecs.umass.edu, {guyu,baoyubin,yuge}@cse.neu.edu.cn

ABSTRACT

Myriad of parameter estimation algorithms can be performed by an Expectation-Maximization (EM) approach. Traditional synchronous frameworks can parallelize these EM algorithms on the cloud to accelerate computation while guaranteeing the convergence. However, expensive synchronization costs pose great challenges for efficiency. Asynchronous solutions have been recently designed to bypass high-cost synchronous barriers but at expense of potentially losing convergence guarantee.

This paper first proposes a flexible synchronous parallel framework (FSP) that provides the capability of synchronous EM algorithms implementations, as well as significantly reduces the barrier cost. Under FSP, every distributed worker can immediately suspend local computation when necessary, to quickly synchronize with each other. That maximizes the time fast workers spend doing useful work, instead of waiting for slow, straggling workers. We then formally prove the algorithm convergence. Further, we analyze how to automatically identify a proper barrier interval to strike a nice balance between reduced synchronization costs and the convergence speed. Empirical results demonstrate that on a broad spectrum of real-world and synthetic datasets, FSP achieves as much as 3x speedup over the up-to-date synchronous solution.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; • **Computer systems organization** → **Cloud computing**;

KEYWORDS

Expectation-Maximization, machine learning, distributed iterative computation, flexible synchronous parallel, straggler

1 INTRODUCTION

Expectation Maximization (EM) [14] is one of the top 10 data mining algorithms [37] used today. Numerous applications including clustering [17, 22], image processing [28], medical research [8], and bioinformatics [38], use EM to estimate parameters. Take K-means clustering [22] as an example. EM refines k clusters (parameters) iteratively, starting from some initial guess. Each iteration consists of an Expectation step (E-step) and a Maximization step (M-step). Based on observed input data points and the current cluster estimation, E-step computes unobserved point-cluster assignment. While, M-step re-estimates clusters to be those minimizing the distance between points and the belonging clusters based on results in E-step (i.e., maximizing the likelihood/objective in general EM applications).

Motivation: Because of applications with massive observed data volume, there is an imperative need for sound and effective EM methodologies. Conventional efforts have made significant advancements, such as mini-batch computation for fast convergence [27, 31, 33] and parallelizing E-step for scalability [21, 36, 41]. However, all of them feature contributions in synchronous systems which suffer from costly global barriers in distributed environments, because of slow, straggling workers (technically, processes or threads). For example, we evaluate the impact of stragglers for K-means using the state-of-the-art synchronous solution [41]. The distributed cluster consists of 16 physical machines. We first submit a job by starting 16 workers, one worker per machine, to process a public dataset HIGGS with 11 million data points and 28 attributes per point. Another job with only one worker is subsequently run on a small synthetic dataset (60 thousand points, 50 attributes per point) produced in a random manner. Obviously, one of the 16 machines run two workers at the same time, and the two workers become stragglers due to resource contention. Compared with the scenario where the second job is not submitted, we find that the runtime of the first job increases from 175 seconds to 246 seconds, leading to 40% performance degradation.

We are aware of some recent works regarding the straggler problem, all of which are far from idle. These techniques include data migration or backup [6, 15, 16, 18] which requires additional network bandwidth, memory and/or compute resources; fine-grained synchronous computation [20, 26] that still suffers from stragglers in the subset of workers; and asynchronous implementation [5, 19, 30] without strict convergence guarantee (i.e., monotonously increasing/decreasing the objective value per iteration) [12].

Hence, a naturally desirable goal for EM algorithms is to pursue a system that (1) can maximally unleash the computational power of fast workers by spending time doing more useful computations instead of waiting for slow, straggling workers at the barrier; (2) enjoys the strict convergence guarantee; and (3) is lightweight — without additional resource requirement.

Problem Analysis: From the convergence perspective, a synchronous barrier is necessary. However, the currently used barrier mechanism cannot cope with our first requirement. This is because it follows a pre-defined synchronous parallel (PSP) design where on

SoCC '17, September 24–27, 2017, Santa Clara, CA, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5028-0/17/09…$15.00
https://doi.org/10.1145/3127479.3128612

every worker, workloads per iteration are assigned prior to computation. A 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. Fast workers thereby block themselves at the barrier.

Our Contributions: In this paper, we explore a path to such a target system mentioned above.

We first challenge the conventional wisdom that EM-based algorithms have to be performed with PSP. We aim to design a new flexible synchronous parallel (FSP) framework that can largely reduce the impact of stragglers. FSP still follows the synchronous model for convergence but no workload is pre-assigned, and hence, no barrier location is pre-established. Instead, it enables a coordinator to actively synchronize all workers if necessary. Once the coordinator initiates a barrier, each worker will pause right after completing an atomic operation and then report intermediate results. Because an atomic operation leads to negligible overhead, the fastest worker pauses at nearly the same time instance with the slowest worker, greatly reducing the waiting time. Besides, we formally prove the convergence of FSP-based EM algorithms.

FSP decouples the global barrier and the policy of processing local workloads on workers. That provides opportunities to design an efficient fault-tolerance solution. Specifically, failed workers can be regarded as stragglers without any progress. That means, before such workers are restarted on standby machines, surviving workers can continue computation without pause. This is useful especially when running EM algorithms on transient resources [32] provided by today's public cloud platforms, such as Amazon EC2 spot instances. Usually, transient resources with lower price may be revoked frequently (failures). Worse, it is very time-consuming to apply for a new spot instance and initialize environments (e.g., installing required Softwares and reloading input data). Thus, it will greatly improve the overall performance if the time before restarted workers return can be utilized.

Note that the flexible barrier in FSP is still not free, and hence seeking a proper barrier interval is very important to balance the synchronization cost and the convergence speed. The flexibility mechanism of FSP, however, forces programmers to either blindly select an interval or experience a long learning curve to understand the internals of underlying engines. To gain optimal performance while insulating programmers from tedious low-level details, we design an adjusting component. It can adaptively and automatically compute a roughly optimal interval. To this end, it constantly collects execution statistics like convergence speed and synchronization costs and employs a recursive policy to identify the final interval.

Our major contributions are summarized as below.

- We propose a new flexible synchronous parallel framework called FSP for fast iterative EM computation. Different from asynchronous and pre-defined synchronous solutions, FSP allows workers to perform a flexible barrier to reduce synchronization costs caused by stragglers, while still providing strict convergence guarantee. FSP also enables an efficient fault-tolerance mechanism.
- We present an interval-adjusting component to identify a proper barrier interval for FSP. By balancing the tradeoff between the synchronization costs and convergence speed, FSP gains optimal performance.
- The resulting prototype system *Flegel* exposes the uniform APIs to users for easily programming various EM algorithms. Although *Flegel* focuses on EM, it can be easily generalized to other machine learning algorithms, such as Stochastic Gradient Descent (SGD).
- Extensive experimental studies explore the performance features of our proposals. We demonstrate that the speedup of *Flegel* compared against the state-of-the-art work is up to 3.

Paper Organization: The remainder of this paper is organized as follows. Section 2 formally introduces EM and Section 3 presents our FSP framework with a formal convergence proof. Section 4 discusses the system design of *Flegel*. Section 5 reports extensive evaluation results. Section 6 highlights the related work and Section 7 finally concludes this work.

2 PRELIMINARIES

This section reviews the EM approach and its efficient variant, followed by a concrete example application.

Full-batch EM Computation [14]: Let X be an observed value of some random variable, typically consisting of n independent data points. X can be decomposed as $\{X_1, ..., X_n\}$. Z is another variable associated with X but unobserved, and $Z = \{Z_1, ..., Z_n\}$. Now we wish to find the maximum log likelihood estimate $L(\theta) = \sum_{i=1}^{n} \log P(x_i|\theta)$ for unknown parameter θ of a model for X and Z. The marginal probability for X_i, $P(x_i|\theta) = \sum_{z_i} P(z_i, x_i|\theta)$, where $P(z_i, x_i|\theta)$ indicates the joint probability for Z_i and X_i parameterized using θ. Usually, it is hard to solve this problem directly since both Z and θ are unknown. However, EM can maximize $L(\theta)$ by starting with some initial guess about $\theta^{(0)}$, and then proceeding to iteratively generate successive estimate, $\theta^{(t)}$, at the t-th iteration ($t = 1, 2, ...$). Each iteration is performed by applying an E-step and a M-step:

- **E-step**: Estimate a distribution $Q_i^{(t)}$ over the range of Z_i for every data point X_i (full-batch), given the currently estimated parameter $\theta^{(t-1)}$. That is, $\forall X_i \in X, Q_i^{(t)}(z_i) = P(z_i|x_i, \theta^{(t-1)})$, subject to $\sum_{z_i} Q_i^{(t)}(z_i) = 1$ and $Q_i^{(t)}(z_i) \geq 0$. In fact, $Q_i^{(t)}$ represents the value of unobserved variable Z_i.
- **M-step**: Update $\theta^{(t)}$ to the θ that can maximize the function $\sum_{i=1}^{n} E_{Q_i^{(t)}}[\log P(z_i, x_i|\theta)]$, where $E_{Q_i^{(t)}}[\cdot]$ denotes expectation with respect to $Q_i^{(t)}$ found in E-step.

Incremental Mini-batch EM Variant: For fast convergence, a mini-batch variant partially implements E-step [27, 31, 33, 41], i.e., only a subset of data points (called a block B_j, s.t. $\cup_j B_j = X$) are computed at an iteration. Note that updating θ depends on all Q_i, but, as discussed above, only some of them are re-calculated in the partial E-step. To eliminate re-computations related to untouched data points, an efficient policy is to compute θ based on a statistics vector $s^{(t)} = \sum_{i=1}^{n} s_i^{(t)}(Z_i, X_i)$, where $s_i^{(t)}(z_i, x_i)$ indicates statistics associated with $Q_i^{(t)}(z_i)$ for X_i. $s^{(t)}$ can be updated incrementally by accumulating the change of $s_i(Z_i, X_i)$, i.e., Δs_i. Further, Neal et al. [27] re-define the goal of mini-batch variants as that, both the E and the M steps try to maximize, or at least increase a new objective function shown in Eq. (1).

$$\begin{aligned} F(Q,\theta) &= \sum_{i=1}^{n} F_i(Q_i,\theta), \textit{where} \\ F_i(Q_i,\theta) &= E_{Q_i}[\log P(z_i,x_i|\theta)] + H(Q_i) \end{aligned} \tag{1}$$

Here $H(\cdot)$ is the entropy of Q_i. Hence, at each iteration, the full-batch EM computation with $L(\theta)$ can be equivalently re-written as Eq. (2) (proved by Theorem 1 in [27]), where $Q_i^{(t)}$ is set to the Q_i that maximizes $F_i(Q_i,\theta)$, given by $Q_i^{(t)}(z_i) = P(z_i|x_i,\theta^{(t-1)})$.

$$\begin{cases} \textbf{E-step: Choose } B_j \textit{ to be updated, and } \forall X_i \in B_j : \\ \qquad \textbf{Set } s_i^{(t)}(Z_i,X_i) = E_{Q_i^{(t)}}[s_i^{(t)}(z_i,x_i)]. \\ \qquad \textbf{Set } \Delta s_i^{(t)} = s_i^{(t)}(Z_i,X_i) - s_i^{(t-1)}(Z_i,X_i). \\ \quad \textbf{Commit } \textit{every } \Delta s_i^{(t)},\ X_i \in B_j. \\ \quad \textbf{Wait } \textit{for newly updated } \theta^{(t)}. \\ \textbf{M-step: Set } s^{(t)} = s^{(t-1)} + \Delta s_i^{(t)},\ X_i \in B_j. \\ \quad \textbf{Set } \theta^{(t)} \textit{ to the } \theta \textit{ that maximizes } F(Q,\theta) \\ \quad \textit{based on } s^{(t)}. \\ \quad \textbf{Broadcast } \theta^{(t)}. \end{cases} \tag{2}$$

K-means example [22, 23]: As a simple EM application in clustering, K-means aims to partition n observed data points in X into k clusters $\theta = \{\theta_1, ..., \theta_k\}$ so as to minimize the objective function: $f = \sum_{j=1}^{k} \sum_{X_i \in \theta_j} ||X_i - \mu_{\theta_j}||$, 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 $\{1, 2, ..., k\}$, indicating to which of k clusters a given observed X_i is supposed to be assigned. Specifically, E-step assigns X_i to the nearest cluster $\theta_{\tilde{j}}$: $Q_i^{(t)}(z_i = j) = 1$, if $j = \tilde{j}$; 0 otherwise ($j \neq \tilde{j}$). The statistics $s^{(t)}$ includes two k-dimensions vectors: S with $S_j = \sum_{X_i \in \theta_j} X_i$, and C with $C_j = |\theta_j|$, $j = 1, 2, ..., k$. Suppose that X_i changes its cluster assignment 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 $\mu_{\theta_j} = \frac{S_j}{C_j}$. Other algorithms, like Fuzzy C-Means (FCM) [17] and Gaussian Mixture Model (GMM) [27, 41], can be implemented in the similar way.

Distributed EM Implementation: For better scalability, data points X can be partitioned into multiple workers (called *data-parallelism*). That can be easily supported by today's *parameter server* systems where the parameter server as a coordinator synchronizes workers (running full-batch or mini-batch E-step) at pre-defined barriers to update θ. These systems provide strict convergence guarantee, but suffer from expensive waiting costs caused by stragglers, — which our new flexible synchronous parallel framework can reduce.

3 FSP FOR EM ALGORITHMS

The novel flexible synchronous parallel (FSP) framework is introduced by discussing three key problems. We first present FSP (Section 3.1) with a theoretical guarantee on algorithm convergence (Section 3.2). Because the flexible barrier can be initiated at any time, we then discuss how to select a reasonable barrier interval in a recursive way (Section 3.3).

3.1 Overview of FSP

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

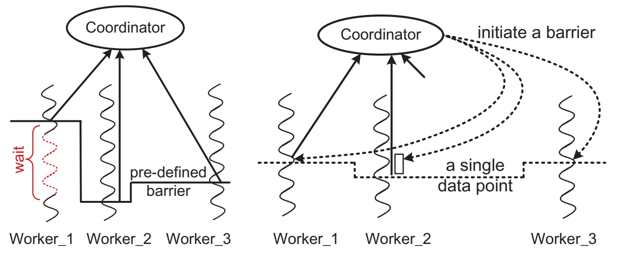
The waiting time in PSP is mainly caused by its built-in pre-defined barriers. Before running EM iterations under PSP, all workers pre-define barrier locations by assigning workloads in E-step per iteration, like processing every local data point (full-batch EM) or some block B_j (mini-batch EM). A global barrier is passively formed when all workers have completed pre-assigned workloads, which is very sensitive to stragglers. Suppose that three workers are used. As demonstrated in Figure 1(a), workers possibly reach their individual pre-defined barriers at different speeds. Fast workers like *Worker_1* thereby block themselves to wait for stragglers like *Worker_2*.

FSP reduces the waiting time by replacing pre-defined barriers with a new flexible barrier mechanism. Specifically, the coordinator can actively initiate a barrier by broadcasting a signal. Once receiving the signal, a worker immediately commits local updates right after computing the current data point, no matter how many data points have been processed since the last barrier. Let $\tilde{B}^{(t)}$ be a subset of data points processed between the (t-1)-th and the t-th iterations. The size of $\tilde{B}^{(t)}$ indicates the workloads per iteration. Different from the pre-defined B_j in Eq. (2) under PSP, $\tilde{B}^{(t)}$ varies with iterations as a new flexible barrier can be initiated at any time. As shown in Figure 1(b), workers first committing updates, such as *Worker_1*, only wait for stragglers, such as *Worker_2*, to process a single data point at most. Usually, computing one data point leads to negligible waiting costs. The coordinator can quickly complete a synchronization operation and then starts the next iteration. As a result, fast workers spend more time to perform computations, instead of waiting for stragglers. Also, the lightweight synchronization barrier can be initiated more frequently so that more up-to-date parameters are visible for workers. Both of the two advantages contribute to boosting the performance.

Now we discuss the detailed behaviors of E-step (at each worker) and M-step (at the coordinator) under FSP. Eq. (3) shows how to perform the incremental mini-batch EM variant (shown in Eq. (2)) using our flexible barrier.

Firstly, like PSP, workers under FSP share the same parameter in E-step. This is because M-step cannot be run until all workers have already committed their local updates at the global barrier. Computations of E-step, in return, can be continued only when receiving newly updated parameters from the coordinator.

Secondly, different from PSP, FSP is essentially required to manage barrier signals to synchronize workers. Costs incurred by signals depend on the specific implementation. An active method is that *Coordinator* actively establishes a network connection with every *Worker*, and then directly modifies the signal status kept in the latter. Detecting the local signal is cheap and hence we can perform it immediately after processing one data point, to minimize the detection latency. However, the broadcasting cost increases with the number



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

Figure 1: Illustration of how to complete a global barrier under (a) the existing PSP framework, and (b) a new FSP framework presented in this paper. The dashed line in (b) indicates a flexible barrier. Besides, the solid arrow line in (a) and (b) indicates committing local updates Δ to the coordinator who performs the parameter update operations.

of *Worker*s. Another method is that *Coordinator* writes the signal into a shared distributed file system, such as HDFS [2], to passively respond the query request from *Worker*s. The broadcasting cost is a constant but *Worker* must carefully select a query frequency because reading data on HDFS incurs network costs. Currently, we employ the active method, since the broadcasting cost can be tolerated in a moderate size cluster based on our tests.

$$\begin{cases} \textbf{E-step: Continuously } \textit{sample } X_i \textit{ and add it into } \tilde{B}^{(t)}: \\ \qquad \textbf{Set } s_i^{(t)}(Z_i, X_i) \textit{ and } \Delta s_i^{(t)} \textit{ as Eq.}(2). \\ \qquad \textbf{If } \textit{a barrier signal is received}: \\ \qquad\quad \textbf{Pause } \textit{local computations}. \\ \qquad\quad \textbf{Commit } \Delta s_i^{(t)},\ X_i \in \tilde{B}^{(t)} \textit{ and wait for } \theta^{(t)}. \\ \textbf{M-step: Initiate } \textit{a barrier signal}. \\ \qquad \textbf{Collect } \textit{every available } \Delta s_i^{(t)} \textit{ and then}: \\ \qquad\quad s^{(t)} = s^{(t-1)} + \Delta s_i^{(t)},\ X_i \in \tilde{B}^{(t)}. \\ \qquad \textbf{Set } \theta^{(t)} \textit{ as Eq.}(2) \textit{ and then broadcast it}. \end{cases} \tag{3}$$

Last but not least, FSP decouples the global barrier and the policy of managing local data on workers. That provides opportunities to design an efficient fault-tolerance mechanism as described below.

• Coordinator failure: In fact, under FSP, any worker can act as a coordinator since parameters anywhere are identical. When the worker where *Coordinator* resides fails, another one can be selected to immediately play the role of new *Coordinator* by notifying surviving workers the change, which tolerates the single point of failure. Meanwhile, the failed worker as a worker failure will be restarted on standby machines as described later.

• Worker failure: When any worker fails, it can be restarted and then continue computation. A failure preprocessing phase is required to schedule the restarted worker, initialize runtime configurations, reload input data, and so on, which is from the point when a failure is detected to the point when the restarted worker starts local computation. During the preprocessing phase, no update is committed. Thus, in PSP, M-step cannot be run because *Coordinator* cannot receive updates from all workers. However, for FSP, we regard the failed worker or the restarted worker as a straggler without any progress. Then EM computations on *Coordinator* and surviving workers can be continued without any pause. EM algorithms can still converge to the correct solution because the restarted worker definitely commits updates after the preprocessing phase. This is urgently required because preprocessing usually tasks too much time, especially when using today's public cloud service platforms. These platforms typically have provisioned spare capacity to meet natural demand fluctuation. It is economically feasible to offer temporarily spare cloud resources (such as Amazon EC2 spot instances, Microsoft Azure Batch, and Google preemptible instances) at a lower price. Yet, these resources are only transiently available as they may suddenly be utilized in case of a load spike. The revocation behaviors (failures) may happen frequently. Worse, it is typically time-consuming to apply for a new spot instance and initialize the configurations (e.g., installing required Softwares). The frequent failures and the expensive restarting cost make it significantly important to continue computation during the preprocessing phase.

3.2 Convergence Analysis

Similar to the traditional PSP framework (Eq. (2)), our FSP framework (Eq. (3)) ensures that each worker shares the same parameter in E-step. This property can be used to prove that FSP enjoys the same convergence guarantee with PSP for various EM algorithms.

THEOREM 3.1. *Incremental mini-batch EM algorithms under the* FSP *framework converge.*

PROOF. Convergence guarantee can be proved by showing that EM computations monotonously increase the objective function value $F(Q, \theta)$ (Eq. (1)). 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 the 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 changes $F_i(Q_i^{(t_j+1)}, \theta^{(t_j)})$, and hence $F(Q, \theta)$, through 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 the parameter $\theta^{(t_j)}$, we can maximize $F_i(Q_i^{(m+1)}, \theta^{(t_j)})$ using a Lagrange multiplier [27], subject to $\sum_{z_i} Q_i^{(m+1)}(z_i) = 1$ and $Q_i^{(m+1)}(z_i) \geq 0$. At such a maximum, we have the unique solution that $\hat{Q}_i^{(m+1)}(z_i) = P(z_i|x_i, \theta^{(t_j)})$, and we indeed use it to initialize $Q_i^{(m+1)}(z_i)$. 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 (see Eq. (3)), 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. [27] formally prove the strict convergence property, i.e., monotonously increasing $F(Q, \theta)$ at each iteration, for centralized EM algorithms. Their proof ignores the discussion on different versions of Q_i and θ, because the sequential model naturally ensures any update result can be immediately available for the subsequent operation. Differently, Theorem 3.1 tells us that a global barrier is essential to guarantee the strict convergence property 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 summation $F(Q, \theta)$, instead of a single $F_i(Q_i, \theta)$. Our FSP framework is thereby different from asynchronous EM variants where parameters are shared in a totally asynchronous [30] or a partially asynchronous [19] way. Besides, each worker in FSP uses a round-robin policy to schedule the update operations among mini-batches. In this way, every data point can be sampled during iterations.

3.3 Determining Barrier Interval

Although a flexible barrier can be initiated at any time without losing the convergence guarantee, the question is that who can trigger this operation and when? We give the answer in this section.

Requesting a barrier by any worker. Theoretically, a worker can cycle through local data points before receiving a barrier signal. Because of unchanged θ within an iteration, computations in multiple full passes of data are redundant, except the first pass. To avoid wasting compute resources, any worker who completes a full pass of data will immediately notify the coordinator, so that the latter can broadcast a barrier signal.

Tuning the barrier interval η by the coordinator. Frequent barrier operations make up-to-date parameters visible and then increase the computation quality, i.e., making great progress per iteration. However, the flexible barrier is still not free and hence the increasing synchronization cost decreases the quantity, i.e., iterations executed per unit time. Identifying a proper barrier interval (η) can balance the tradeoff between quality and quantity.

A proper η is determined by the coordinator in a recursive way. The main idea is to compare the change of the objective function value per unit time under the current interval η and a smaller $\frac{\eta}{\lambda}$, respectively. Here λ is the adjusting step size and typically set to 2 (user can specify another value). If the objective value changes faster under $\frac{\eta}{\lambda}$ when compared with η, then $\eta = \frac{\eta}{\lambda}$. This process is done recursively until the comparison result is reversed.

A problem now is how to efficiently make the comparison. If we run the EM application under η and $\frac{\eta}{\lambda}$ to respectively get the accurate changes, the tuning runtime will be so large that we cannot tolerate it. A feasible solution is to get one accurately, and then estimate the other. Usually, over a short period of time, if η keeps invariant, the objective value is predictable because it varies roughly linearly with elapsed time. Thus, the change under η is chosen as the estimate variable. For the accurate one, after finishing the current iteration, we use $\frac{\eta}{\lambda}$ to perform the subsequent λ iterations, and then an objective value change $\tilde{p}_{[\frac{\eta}{\lambda}]}$ is accurately computed. $\tilde{p}_{[\frac{\eta}{\lambda}]}$ is achieved during the runtime $T = \frac{\eta}{\lambda} \cdot \lambda + \varphi \cdot (\lambda - 1)$, because $(\lambda - 1)$ barriers with overhead φ per barrier are incurred. Now we estimate the change under η during T. Let p be the objective value change in the most recent iteration using η. The estimate change is $\tilde{p}_{[\eta]} = T \cdot \frac{p}{\eta}$.

Based on the comparison, if $\tilde{p}_{[\eta]} < \tilde{p}_{[\frac{\eta}{\lambda}]}$, $\frac{\eta}{\lambda}$ is better than η, and can be further decreased in remaining computations until $\tilde{p}_{[\eta]} \geq \tilde{p}_{[\frac{\eta}{\lambda}]}$. Now the recursive adjusting process terminates and takes the average of $\{\eta, \frac{\eta}{\lambda}\}$ as the optimal interval[1]. Alg. 1 summarizes the adjusting process. In fact, a full-batch iteration will be run to initialize the global statistics s before incremental computations shown in Eq. (3). The full-batch runtime as the largest interval can be used to initialize the input value of η in Alg. 1.

```
Algorithm 1: interAdjusting algorithm
Input  : Current interval $\eta$, adjusting step size $\lambda$, objective
         value change $p$ at one iteration and synchronization
         cost $\varphi$
Output : A proper barrier interval
1 Get accurate $\tilde{p}_{[\frac{\eta}{\lambda}]}$ by running $\lambda$ iterations with $\frac{\eta}{\lambda}$;
2 Compute $\tilde{p}_{[\eta]}$ based on $p$ and $\varphi$;
3 if $\tilde{p}_{[\eta]} < \tilde{p}_{[\frac{\eta}{\lambda}]}$ then
4     interAdjusting($\frac{\eta}{\lambda}, \lambda, \frac{\tilde{p}_{[\frac{\eta}{\lambda}]}}{\lambda}, \varphi$);
5 else
6     return $\frac{\eta + \frac{\eta}{\lambda}}{2}$;
```

Prior experimental investigations [33] reveal that for mini-batch EM computations, the batch size that yields the best speedup during initial iterations is roughly the optimal size in the whole iterations. Motivated by this, we can fix η after identifying a proper value, if the compute environment is not changed in subsequent iterations. When changes are detected, the coordinator can seek another η by invoking Alg. 1 again.

4 FLEGEL: A FSP-BASED SYSTEM

Now we present *Flegel*, a memory-based distributed implementation of our FSP framework (Section 4.1). It exposes high-level APIs to users for easily implementing various EM algorithms (Section 4.2), and employs a new termination check mechanism to automatically detect the algorithm convergence (Section 4.3). Further, *Flegel* can be generalized to other machine learning algorithms such as Stochastic Gradient Descent (SGD) (Section 4.4).

[1]Although a further recursive comparison between η and $\frac{\eta + \frac{\eta}{\lambda}}{2}$ computes a more proper interval, the performance improvement is not significant based on our tests. Thus, we simply terminate the recursion to reduce the tuning cost.

4.1 Design of Flegel

Architecture: Figure 2 gives the overview architecture of *Flegel*, which employs a widely used Master-Slave design consisting of a Master machine and K Slave machines. Master is in charge of slaves, including monitoring their healthy status and being aware of the load variation on the cloud. It also responds concurrent requests of running EM algorithms submitted by users, and then manages jobs, such as notifying a job to tune its barrier interval η.

The execution of one job is divided into several workers which are scheduled onto different slaves. Before computations, input data points are also divided into partitions, each of which is kept on a worker as data table so that they can be processed in parallel. Data table consists of a series of triples (point *id*, value Q, and property information *info.*). Currently, we use a range partition policy [1]. Typically, one of these workers is selected as the coordinator to maintain parameters, while others store a copy. The coordinator updates parameters and then synchronizes all local copies. In particular, for algorithms with millions of parameters, one or more workers on physically separate machines can be selected as the coordinator for efficiently updating parameters.

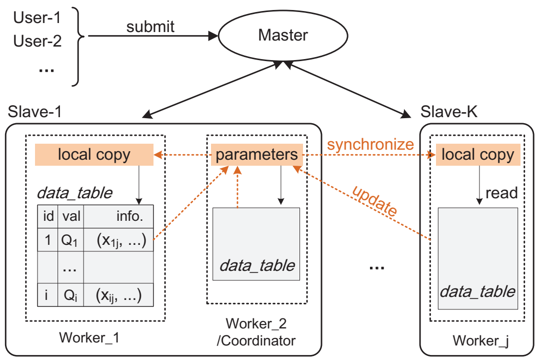


Figure 2: Architecture of *Flegel*

Execution of an EM algorithm on *Flegel*: Algorithm 2 and Algorithm 3 respectively summarize the execution of *Coordinator* and *Worker*. Once *Worker* receives θ from *Coordinator*, the EM computation starts (Line 6 in Alg. 2 and Line 4 in Alg. 3). First of all, a full-batch iteration is performed to initialize data point values (i.e., Q_i) using θ^0 (Lines 12-13 in Alg. 2 and Lines 4-8 in Alg. 3), so that in subsequent computations, the efficient incremental iteration can be run (Lines 6-14 in Alg. 2 and Lines 10-27 in Alg. 3). Note that Alg. 2 leaves out several details for synchronous barrier interval (η) selection (Alg. 1), as well as details specific to detecting convergence, i.e., *terminationCheck*() (Section 4.3).

Lastly, in Alg. 3, we show how to detect a full pass through local data so that the worker can trigger a global barrier. Towards this end, each worker marks the beginning position of a new iteration, i.e., resetting a variable w to zero (Lines 12 and 26). Note, that if the marked position is reached again before the next barrier happens, then a full pass of data is done (Line 14).

Algorithm 2: *Coordinator* algorithm

```
1  Initialize parameter $\theta^{(0)}$;
2  Initialize synchronous barrier interval $\eta$ to $+\infty$;
3  Initialize iteration counter $t$ to 1;
4  Initialize statistics vector $s^{(0)}$ to $\mathbf{0}$;
5  while terminationCheck() is False do
6      Broadcast $\theta^{(t-1)}$;
7      //Perform full-batch computations under PSP if $t$=1;
8      if $t > 1$ then
9          Update $\eta$ if necessary;
10         Sleep for $\eta$ milliseconds;
11         Broadcast signal_barrier to initiate a barrier;
12     Aggregate statistics from workers to update $s^{(t)}$;
13     Compute $\theta^{(t)}$ based on $s^{(t)}$ and $\theta^{(t-1)}$;
14     $t \leftarrow t$+1;
15 Broadcast signal_termination to terminate computations;
```

4.2 APIs

We illustrate the programming APIs used in *Flegel* in Figure 3. The function of *initParameter*() describes about how to initialize parameter $\theta^{(0)}$ as the input of the 1st iteration. *updateDataPoint*() is responsible for computing a given data point based on parameters updated in the previous iteration, and then returning an aggregator containing the change of local statistics which can be immediately accumulated by invoking *aggregate*(). *aggregate*() will be called again at the flexible barrier to aggregate reports from all *Worker*s to get global statistics. After that, we can update parameters using *updateParameter*() specified by users. The *terminationCheck*() function tells *Flegel* about whether or not to terminate iterations.

```
// 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);
```

Figure 3: APIs provided by *Flegel*

4.3 Termination Check

The default termination check implementation is to set the maximum number of iterations, but users can override this function to meet their own requirements. One choice is to set an ideal objective value prior to computation. Then the convergence point is reached if the newly computed objective value based on *aggregator* is above or below the pre-defined target.

Another widely used alternative is first computing the difference between two consecutive objective values, and then terminating iterations if the difference is less than a given threshold. An important problem here is that the barrier interval in FSP is unknown prior to

Algorithm 3: *Worker* algorithm

```
FIFO queue $dataQ \leftarrow loadData()$;
Initialize the change of local statistics $\Delta s$ to **0**;
//Perform full-batch computations under PSP;
Get $\theta$ from Coordinator;
foreach $X_i$ in dataQ do
    Compute $\Delta s_i$ based on $X_i$ and $\theta$;
    $\Delta s \leftarrow \Delta s + \Delta s_i$;
Commit $\Delta s$ and then reset it to **0**;
//Perform mini-batch computations under FSP;
Get $\theta$ from Coordinator;
Initialize signal to signal_continue;
Initialize the mark $w$ to 0;
while signal is not signal_termination do
    if $w$ equals the size of dataQ then
        Notify Coordinator to initiate a barrier;
        Wait for a new signal;
    else
        Remove $X_i$ from the head of dataQ;
        $w \leftarrow w + 1$;
        Compute $\Delta s_i$ based on $X_i$ and $\theta$;
        $\Delta s \leftarrow \Delta s + \Delta s_i$;
        Add $X_i$ to the tail of dataQ;
    Update signal if a new one has already been received;
    if signal is signal_barrier then
        Commit $\Delta s$ and then reset it to **0**;
        Reset $w$ to 0;
        Get $\theta$ from Coordinator;
Dump computation results;
```

running a job. An extremely short interval, of course, leads to a small variation and hence a false-positive check result, — which triggers an abnormal termination. We alleviate this problem by decoupling termination check and parameter update. The check interval is set to the runtime of performing a full pass computation over data on the slowest worker, regardless of the update interval η. Hence, a big difference can be provided for the *terminationCheck* function.

4.4 More Machine Learning Algorithms

Besides EM, *Flegel* also supports other machine learning algorithms. The general goal of machine learning is to refine a model (θ) by iterating over input data (X), so as to perform future predictions over new data. The refining quality is measured by a loss function $f(\theta) = \frac{1}{n}\sum_{i=1}^{n} F_i(x_i;\theta)$, where $n = |X|$, $x_i \in X$, and $F_i(x_i;\theta)$ indicates the loss w.r.t. x_i. This is achieved using a stochastic optimization policy with two steps at each iteration: 1) computing a stochastic gradient ∇F_i (resembling the distribution of unobserved variable Q_i), and 2) updating parameter θ based on ∇F_i. Taking Stochastic Gradient Descent (SGD) as an example, its mini-batch variant updates θ by

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

where α is the *learning rate* or *step size* and B is a batch of data points sampled between two inserted flexible barriers in *Flegel*. Similar to the EM implementation, the two steps can be performed in E-step and M-step, respectively.

Logistic Regression (LR): We next use LR to show that *Flegel* is also applicable to SGD. Each data point $\bar{x}_i \in X$ contains d attributes with one additional value y_i, that is, $\bar{x}_i = \{x_i, y_i\} = \{\{x_{i1}, x_{i2}, ..., x_{id}\}, y_i\}$. Any y_i has a boolean domain $\{0, 1\}$. LR on input data X returns a function P parameterized using a vector θ, which predicts $y_i = 1$ with probability

$$P(x_i) = \frac{1}{1 + exp(-x_i^T \theta)}$$

The optimal θ minimizes the loss function

$$F(x_i;\theta) = (y_i - 1)\log(1 - P(x_i)) - y_i \log P(x_i),$$

and each dimension $\theta_k^{(t)}$ of θ can be refined iteratively by

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

In *Flegel*, X is partitioned onto workers and stored in *data table* (shown in Figure 2), while the weight vector θ is the shared parameter kept in the coordinator. Specifically, *val* in *data table* is extended to store the stochastic gradient vector ∇F_i and the y_i value, i.e., (∇F_i, y_i). At the very beginning of computation, *initParameter*() first gives some initial guess of θ. *updateDataPoint*() then computes ∇F_i by $\frac{\partial}{\partial \theta_k} F(x_i;\theta^{(t-1)})$ given input data point $\bar{x}_i$ and the parameter θ. All gradients are accumulated by *aggregate*() and finally averaged to update θ in *updateParameter*().

In the similar way, *Flegel* also supports other SGD based applications, such as Matrix Factorization [40] and Support Vector Machine (SVM) [11]. In future, we will also investigate whether our proposal can be used in iterative graph processing systems [24, 34, 39].

5 PERFORMANCE STUDIES

We conduct extensive experiments in this section. The general experimental details are given below:

Frameworks for comparison: We analyze the performance of our basic Flexible Synchronous Parallel framework (FSP) where the barrier interval is simply fixed as runtime of the first full-batch iteration, and its improved version (smartFSP) which smartly computes a proper interval. We compare the two solutions against traditional Pre-defined Synchronous Parallel framework (PSP) which performs full-batch computation on every iteration, as well as the up-to-date mini-batch variant (miniPSP) [41] with an automatically computed batch size. All frameworks are implemented on *Flegel* based on Java, in order to make an end-to-end performance comparison.

Experimental cluster: Our cluster consists of 32 physical machines (slaves) with one additional master machine connected by a Gigabit Ethernet switch. Each machine is configured with 8 cores (Intel Core i3-2100, 3.1GHz) and 8GB of RAM, running on top of Red Hat Enterprise Linux 6.0. Unless otherwise specified, 16 workers are used for a given EM job, and further, the fair worker-scheduler in *Flegel* ensures that one machine runs one worker at the same time, to avoid possible resource contention.

EM algorithms and datasets: We test three representative EM applications, namely, K-means, Fuzzy C-Means (FCM) [17], and

Gaussian Mixture Model (GMM) [27, 41] to explore the performance features of all tested frameworks. K-means and FCM are tested over publicly available datasets MASS[2] and HIGGS[3] from UCI Machine Learning Repository, while GMM is run on synthetic datasets generated in a random manner, as shown in Table 1.

Table 1: Datasets Summary

Algorithms	Datasets	Points	Dimensions
K-means/FCM	MASS	7,000,000	27
	HIGGS	11,000,000	28
GMM	Synth-D	5,000,000	40
	Synth-P	10,000,000	20

Because K-means has been introduced in Section 2, now we present the detailed implementation of FCM and GMM.

• FCM: FCM as a "soft" clustering variant, allows X_i to be assigned into total k clusters with a k-dimensions weight vector w_i, where

$$w_{ij} = \frac{1}{\sum_{l=1}^{k}(\frac{||X_i-\mu_{\theta_j}||}{||X_i-\mu_{\theta_l}||})^{\frac{2}{q-1}}},$$

and q is a user-specified fuzzy factor ($q > 1$ and here we set it to 1.1). The objective function is given by:

$$f = \sum_{j=1}^{k}\sum_{i=1}^{n} w_{ij}^{q}||X_i - \mu_{\theta_j}||^2.$$

Now, in E-step, $Q_i^{(t)}(z_i = j) = w_{ij}$. Like K-means, the two statistic vectors S and C are defined as $S_j = \sum_{i=1}^{n} w_{ij}^{q} X_i$, and $C_j = \sum_{i=1}^{n} w_{ij}^{q}$. When w_{ij} is changed into $w_{ij}^{'}$, S and C are updated incrementally: $S_j = S_j + \big((w_{ij}^{'})^q - (w_{ij})^q\big)X_i$; $C_j = C_j + \big((w_{ij}^{'})^q - (w_{ij})^q\big)$.

• GMM: The goal of GMM is to specify how likely a given data point X_i is generated from the j-th Gaussian distribution with the mean c_j and covariance matrix Σ_j, where $j \in \{1, 2, ..., k\}$. $\theta_j = (c_j, \Sigma_j)$. EM maximizes the objective function:

$$f = \frac{1}{n}\sum_{i=1}^{n}\log\big(\sum_{j=1}^{k} w_j P(x_i|\theta_j)\big),$$

where w_j stands for a weight value. $P(x_i|\theta_j)$ indicates the probability of generating X_i from the j-th Gaussian distribution. Let d be the cardinality of X_i . Then P can be computed below:

$$P(x_i|\theta_j) = (2\pi)^{-\frac{d}{2}} \cdot |\Sigma_j|^{-\frac{1}{2}} \cdot e^{-\frac{1}{2}(x_i-c_j)^T \cdot \Sigma_j^{-1} \cdot (x_i-c_j)}.$$

The normalized probability value γ_{ij} is given by:

$$\gamma_{ij} = \frac{w_j P(x_i|\theta_j)}{\sum_{l=1}^{k} w_l P(x_i|\theta_l)}.$$

The statistics include three k-dimensions vectors, $\hat{S}$, S and C, and are computed below:

$$\hat{S}_j = \sum_{i=1}^{n}\gamma_{ij}X_i^2; S_j = \sum_{i=1}^{n}\gamma_{ij}X_i; C_j = \sum_{i=1}^{n}\gamma_{ij}.$$

[2]http://archive.ics.uci.edu/ml/datasets/HEPMASS, real dataset

[3]http://archive.ics.uci.edu/ml/datasets/HIGGS, produced by Monte Carlo simulations

The workload of E-step is to first compute the new value of γ_{ij}, $\gamma_{ij}^{'}$. Let $\delta = (\gamma_{ij}^{'} - \gamma_{ij})$, E-step then updates the statistics by: $\hat{S}_j = \hat{S}_j + \delta X_i^2$, $S_j = S_j + \delta X_i$, and $C_j = C_j + \delta$. Finally, M-step recomputes parameters by $w_j = \frac{C_j}{n}$, $c_j = \frac{S_j}{C_j}$, and $\Sigma_j = \frac{\hat{S}_j}{C_j} - \frac{S_j^2}{C_j^2}$.

The testing configurations of EM algorithms under PSP are given below. (1) For K-means, we select the top-100 (k=100) data points in ascending order of ids as initial clusters and then run 100 iterations. (2) For FCM and GMM, because of the heavy workload in E-step, we decrease the number of iterations to 50, and set k to 10 and 5, respectively. In particular, the input clusters of FCM are initialized using the same way with K-means. However, for GMM, random clusters are used to avoid the divide-by-zero exception. miniPSP, FSP and smartFSP, employ the same initialization policy with PSP, but a different termination check method. Specifically, they terminate iterations when f is over (GMM) or below (for K-means and FCM) the corresponding target achieved in PSP. Hence, any EM algorithm under all compared frameworks will converge to the same solution, so that we can make a fair comparison.

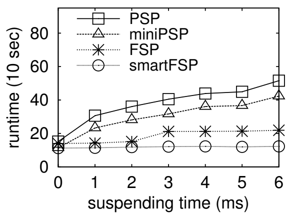
Evaluation metrics: Two metrics are evaluated in experiments, *runtime* and *objective function value*. *Runtime* is defined as the elapsed time from the point when computation starts to the point when an algorithm converges. The overheads of loading data and dumping results are excluded since they are the same for all compared frameworks. *Objective function value*, i.e., f, has been given when introducing tested algorithms. In particular, f needs to be minimized for K-means and FCM, but maximized for GMM.

Experiment design: We experiment with injected and naturally-occurring straggler patterns. Generally, stragglers are caused by at one or more factors: (1) different hardware configurations; (2) imbalanced data partitioning; (3) background operating system activities; (4) garbage collection; (5) failure recovery; and (6) imbalanced distribution of workers among physical machines. Users encounter the last scenario when running multiple jobs concurrently. The worker-scheduler fairly schedules workers of a job based on available resources per machine. However, one job (and its workers) may terminate at any time due to convergence. That leads to an imbalanced distribution of workers for remaining running jobs. In particular, workers on congested machines become stragglers.

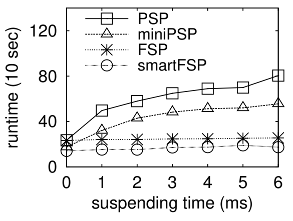
It is hard to accurately evaluate the impact of these real-world factors on runtime. Thus, like Ref. [18], we first assume that *Flegel* suspends computing threads on user-specified workers for τ milliseconds every 1,000 data points processed, so that we can simulate the slow, straggling workers. In this way, a deep performance analysis is made (Sections 5.1-5.4). We also test all frameworks in two real scenarios: failure recovery and imbalanced worker distribution (Section 5.5). The former also demonstrates that *Flegel* can efficiently tolerate failures. Finally, we test the scalability of *Flegel* (Section 5.6) and compare our optimal solution smartFSP against the asynchronous framework (Section 5.7).

5.1 Impact of the Straggling Parameter τ

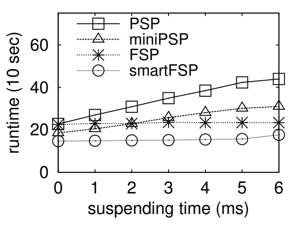
We first evaluate the performance scalability by varying how much a straggler slows down the overall progress, i.e., varying τ from 0 to 6 milliseconds. Figure 4 shows the runtime of compared frameworks for three algorithms over different datasets. In all testing cases, smartFSP always works best. In particular, it outperforms PSP, miniPSP, and FSP by up to 4.63X (from 1.34X), 3.45X (from 1.02X), and 1.79X (from 1.18X), respectively. That mainly stems from the flexible barrier mechanism where fast workers do not wait for stragglers, and the smartly determined barrier interval.



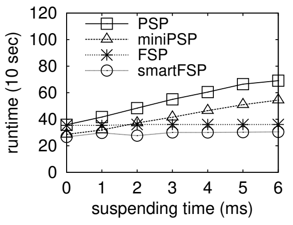
(a) K-means on MASS



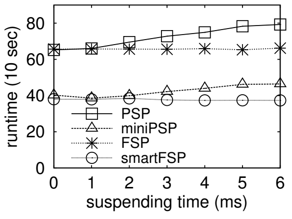
(b) K-means on HIGGS



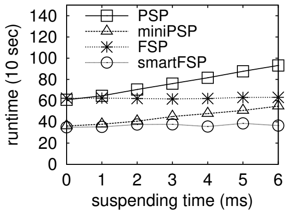
(c) FCM on MASS



(d) FCM on HIGGS



(e) GMM on Synth-D



(f) GMM on Synth-P

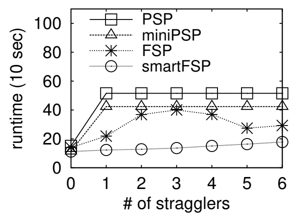
Figure 4: Runtime of different frameworks when varying how much a straggler slows down the overall progress (quantified by $\tau \cdot \frac{\#Points}{1000 \cdot \#Worker}$). X-axis indicates the suspending time per 1000 data points, i.e., τ (millisecond).

PSP and miniPSP essentially get performance degradation because of increasing waiting costs caused by the slowest worker. However, benefitting from frequently running M-step, miniPSP converges faster than PSP, and even beats FSP, particularly when τ is small (FCM in sub-figures (c)-(d)) and/or underlying point data updates are very time-consuming (GMM in sub-figures (e)-(f)).

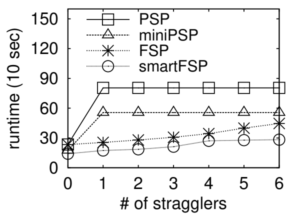
5.2 Impact of Number of Stragglers

With fixed $\tau = 6$ milliseconds, we explore the performance variation when varying the number of stragglers.

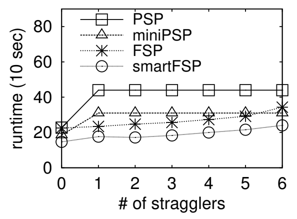
As plotted in Figure 5, the performance gap between FSP and PSP (also smartFSP and miniPSP) narrows down with more stragglers. Specifically, we see that the algorithm performance under PSP/miniPSP does not change significantly, because the synchronization cost is always dominated by the slowest worker, regardless of how many workers are slow (#≥1). On the other hand, the convergence runtime of FSP/smartFSP increases with more stragglers, because updates on stragglers are not so many as those on normal, fast workers. Even so, the speedup of our smartFSP compared with PSP/miniPSP is still up to 2.90X/2.38X in our worst testing case ($\frac{6}{16}\% = 37.5\%$ workers are slow).



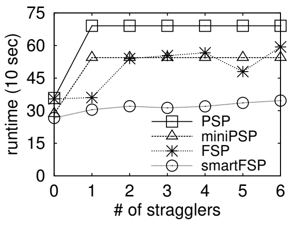
(a) K-means on MASS



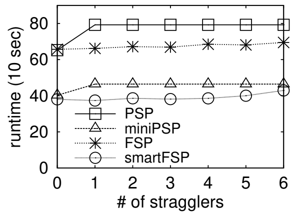
(b) K-means on HIGGS



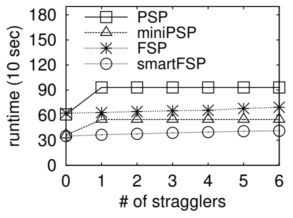
(c) FCM on MASS



(d) FCM on HIGGS



(e) GMM on Synth-D



(f) GMM on Synth-P

Figure 5: Runtime of different frameworks when varying the number of stragglers ($\tau = 6$ milliseconds).

One observation we can make is that the performance of FSP drops significantly in some cases, such as Figure 5(a) and Figure 5(d), when compared with Figure 4(a) and Figure 4(d), respectively. This may be because intermediate results from data points residing on stragglers are greatly important for convergence. Reducing the number of updates on these points forces K-means/FCM to make a long detour to converge. smartFSP, however, is able to largely alleviate the runtime degradation via its interval adjusting component — which takes convergence speed into consideration when computing the proper interval.

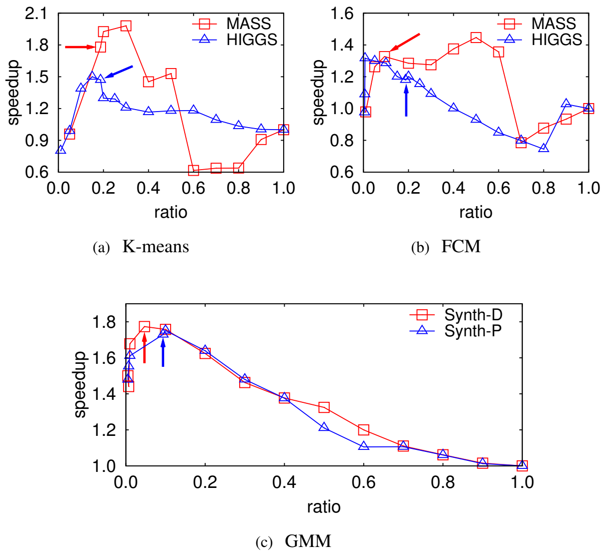
5.3 Verifying Optimal Barrier Interval

Now we design experiments to answer the following questions:

- Effectiveness: how likely can smartFSP seek the optimal barrier interval η?
- Efficiency: how much time does smartFSP take to seek η that is optimal or close to optimal?

All tests are run with $\tau = 6$ milliseconds and 1 straggler.

Effectiveness: To find a real optimal interval, we repeatedly run an EM algorithm and we manually set the barrier interval every time. Figure 6 reports the speedup normalized to FSP against the ratio of manually chosen interval η in smartFSP to the runtime of a full-batch iteration.



(a) K-means

(b) FCM

(c) GMM

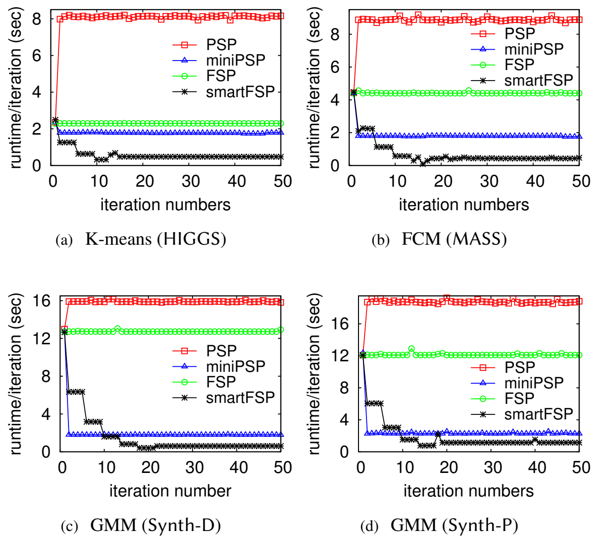
Figure 6: Effectiveness of seeking optimal η. X-axis indicates the ratio of η to the runtime of a full-batch iteration. Arrows point out the optimal intervals computed by smartFSP**.**

With the ratio being increased, we can easily find that the speedup first increases rapidly and then gradually decreases in most cases. This can be explained by the tradeoff between expensive synchronization costs incurred by frequent barriers (small interval) and slow convergence speed due to infrequent barriers (large interval). As shown in Figure 6, smartFSP can roughly hit the "sweet spot" by its online adjusting component. Note that the speedup curve varies with the specific EM algorithm and dataset. Thus, it is impossible to get an one-fit-all solution to find the "sweet spots" by offline analysis used in Ref. [41].

Efficiency: To understand why smartFSP performs better, we further study the efficiency of identifying the optimal η by showing the variation of runtime per iteration. Figure 7 shows the results of some testing cases. We omit other cases for brevity since they exhibit the similar performance. All sub-figures demonstrate that smartFSP can quickly identify a proper η in early iterations.

5.4 Empirical Validation of Convergence

Besides a formal convergence proof in Section 3.2, now we empirically validate that our proposals are applicable to tested algorithms.



(a) K-means (HIGGS)

(b) FCM (MASS)

(c) GMM (Synth-D)

(d) GMM (Synth-P)

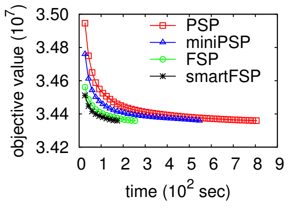
Figure 7: Efficiency of identifying the roughly optimal η.

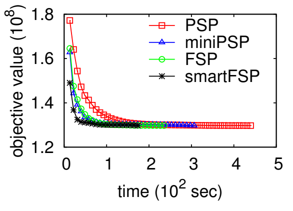
We first test the correctness and then analyze the distribution of contributions made to convergence by each worker.

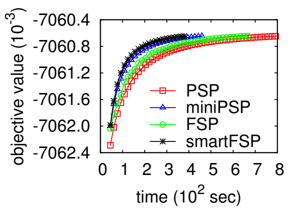
Correctness: The correctness is validated by giving the objective value variation during iterations. Using the same settings in Section 5.3, Figure 8 shows objective value vs. time plots achieved by continuously monitoring the computation progress at sampled time instances. Clearly, all sub-figures reveal that our proposals monotonously decrease or increase the objective value, like traditional PSP and miniPSP. Hence, algorithms under our *Flegel* can converge to the correct solution, and more importantly, they exhibit a faster convergence speed than existing implementations.

Data bias w.r.t. convergence: We further show that in *Flegel*, all workers evenly contribute to refining the objective function value, even though some of them are straggling. Towards this end, after an algorithm converges, we report two metrics for each worker: 1) the number of completed full passes over local data (*passes*), indicating the computation speed; and 2) the total accumulated changes (*delta*) with respect to the objective value, indicating the contribution made by this worker to convergence. We test all of the three algorithms using two different scenarios w/o and w/ injected stragglers ($\tau = 6$ milliseconds and the number of stragglers is 6), so as to demonstrate the metric variation.

Sub-figures (a) and (b) in Figures 9-11 respectively show the comparison results. We observe that when changing τ from 0 to 6, although different workers proceed at different speeds, the objective function value does not skew in the favor of fast workers that complete more passes than stragglers. This is not surprising because data points on stragglers are always updated according to up-to-date parameters pushed by the coordinator. That means although the number of updates (or passes) decreases, the contribution per update largely increases. As a result, the total contribution (*delta*) from stragglers is still significant.


(a) K-means (HIGGS)


(b) FCM (MASS)


(c) GMM (Synth-D)

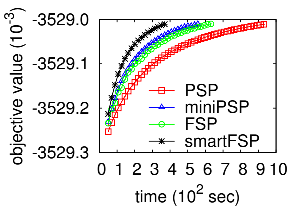
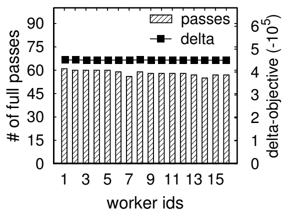

(d) GMM (Synth-P)

Figure 8: Convergence of FSP and smartFSP


(a) $\tau = 0$ ms

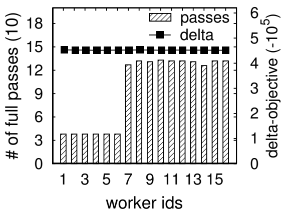
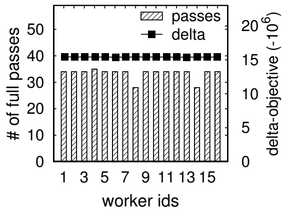

(b) $\tau = 6$ ms

Figure 9: K-means on HIGGS


(a) $\tau = 0$ ms

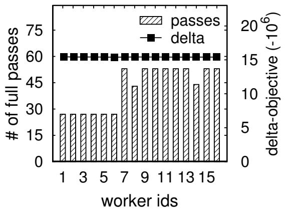
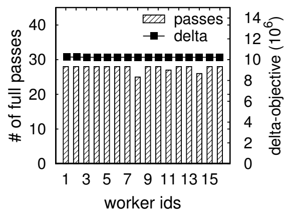

(b) $\tau = 6$ ms

Figure 10: FCM on MASS

5.5 Evaluation in Real-world Environments

As an indication of how different frameworks scale in real environments, Figure 12 depicts the impact of failure recovery and imbalanced worker distribution. We show that FSP can offer a comparable performance to miniPSP, but the former does not require complex offline analysis to compute a mini-batch size. Further, smartFSP always has superior performance to all competitors.


(a) $\tau = 0$ ms

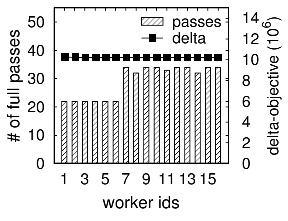

(b) $\tau = 6$ ms

Figure 11: GMM on Synth-P

Failure recovery (case-1): In this suite of experiments, there are total 16 workers. After running an EM algorithm, we manually kill one of them at the 10th and the 30th iterations to simulate two worker failures. Over the period of failure preprocessing, both PSP and miniPSP suspend computation. However, FSP and smartFSP can continue computation without any pause. Note that for all frameworks, failed workers are restarted and then continue local computation after the preprocessing. Sub-figures (a), (c), and (e), report the convergence runtime. miniPSP beats FSP for GMM because the fast convergence speed caused by fine-grained barriers offsets expensive synchronization costs. However, our improved smartFSP runs up to 1.88 times faster than miniPSP.

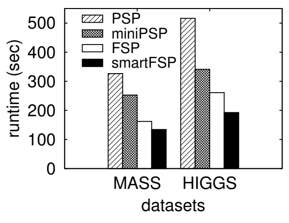
Imbalanced worker distribution (case-2): To simulate the imbalanced distribution, we submit two jobs to *Flegel* concurrently using 16 physical machines: Job-1 with 16 workers and Job-2 with only 1 worker. Because of the fair scheduler, one machine runs two workers, while others run a single one. In particular, Job-2 runs K-means on a synthetic dataset (60 thousand points, 50 attributes per point) produced in a random manner. Sub-figures (b), (d), and (f) report the performance of Job-1 for different algorithms. In comparison with miniPSP, the flexible barrier in FSP brings marginal benefit, and even slight performance degradation due to coarse-grained barriers. While, smartFSP beats miniPSP by a factor of 1.58 at most.

5.6 Scalability

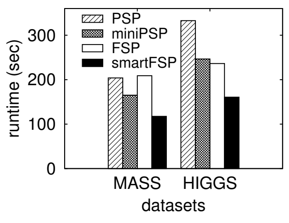
We test the scalability of different frameworks with $\tau = 6ms$ and 1 straggler. For K-Means and FCM over public datasets, we vary the total number of workers from 8 to 32. Figure 13 shows that the runtime decreases gradually. This makes sense because the size of local data points on each worker shrinks linearly when increasing the number of workers. For GMM on synthetic datasets, we first run it on Synth-D using different cluster sizes (Figure 14(a)). Then we generate 4 datasets from 5 million points to 20 million points using the same dimension number 40. Figure 14(b) demonstrates the linear performance change of GMM on a cluster with 16 workers.

5.7 Comparison with Asynchronous Framework

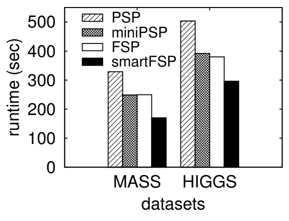
We finally implement asynchronous EM computation on *Flegel* by completely removing the global barrier [10, 13]. We then perform an end-to-end comparison between it and our smartFSP, since the latter has been shown to give better performance than other synchronous frameworks in previous experiments. Figures 15-17 plot objective value against time for the three tested algorithms by setting $\tau = 0$ ms



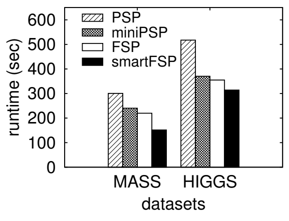
(a) K-means (case-1)



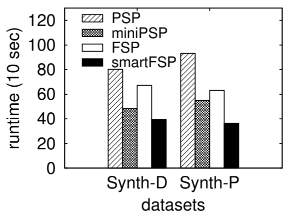
(b) K-means (case-2)



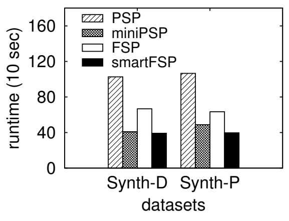
(c) FCM (case-1)



(d) FCM (case-2)

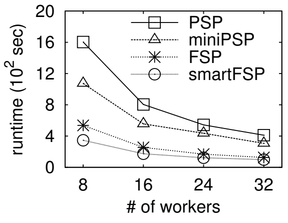


(e) GMM (case-1)

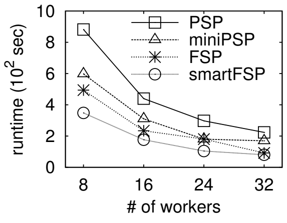


(f) GMM (case-2)

Figure 12: Performance in real-world environments.

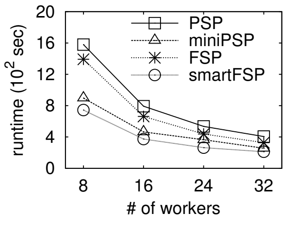


(a) K-means on HIGGS

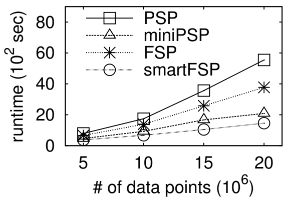


(b) FCM on MASS

Figure 13: Scalability (K-Means and FCM)

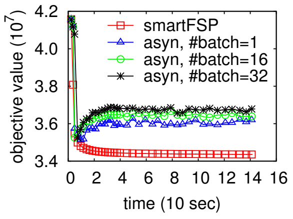


(a) Scaling with number of workers

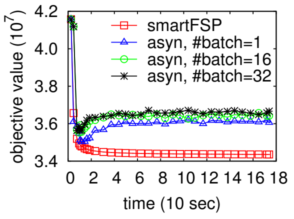


(b) Scaling with data size

Figure 14: Scalability (GMM)

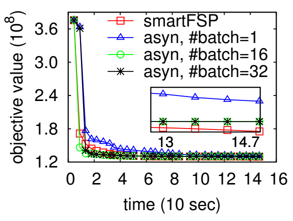


(a) τ = 0 ms

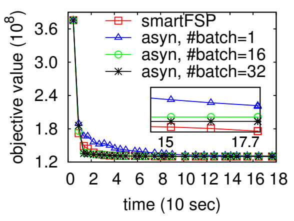


(b) τ = 6 ms

Figure 15: Kmeans on HIGGS

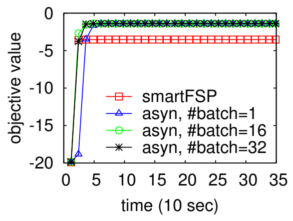


(a) τ = 0 ms

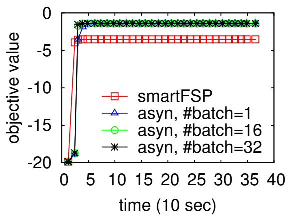


(b) τ = 6 ms

Figure 16: FCM on MASS



(a) τ = 0 ms



(b) τ = 6 ms

Figure 17: GMM on Synth-P

and 6 ms (with one straggler) respectively. Note that smartFSP can automatically identify a proper barrier interval to support mini-batch computation. While, for asynchronous EM, we manually set the number of batches on each worker, i.e., changing the batch size, to explore the performance features.

There is no doubt that the asynchronous framework largely mitigates synchronization costs, but this benefit usually cannot translate to a fast convergence speed with the same output parameter (e.g., K-means and FCM in Figures 15 and 16). This is because asynchronous update on the global parameter makes some workers use stale local copies, which incurs errors. It is extremely difficult to analyze the convergence of asynchronous EM algorithms prior to computation, and indeed such implementations can readily diverge (like K-means in Figure 15). Thus, the asynchronous framework is not a preferred solution even though it works well in some cases (e.g., GMM shown in Figure 17). In contrast, our proposals provide enough flexibility for EM-based algorithms to be highly efficient.

Another observation we find is that for asynchronous K-means, 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 is also a big challenge for users.

6 RELATED WORK

Many representative techniques have been developed to optimize centralized EM algorithms. Today's distributed computation systems also enable scalable EM implementation efforts in the big data era. Below, we review these existing works from the two perspectives to highlight our contributions.

Centralized EM optimizations: Conventional efforts have made significant advancements in three different but complementary directions. The most important branch is to transform full-batch update into mini-batch variant for efficiency [27, 31, 33]. Some important problems have been deeply analyzed, such as convergence guarantee and incremental implementation. In particular, our convergence proof is based on the work of Neal et al. [27], but goes further. Specifically, we prove that in distributed environments, the convergence can be guaranteed by sharing the same parameters among workers. Pre-assigning workerloads to workers is thereby unnecessary. The other two branches include carefully selecting initial input [9] and priority computations (e.g., prioritizing data points in E-step [33] or parameters in M-step [14, 25]). We can easily plug all of the three techniques into our system *Flegel*.

Distributed EM implementations and the straggler problem: The input training data are rapidly growing in size. In order to efficiently run machine learning algorithms including EM, there are a flurry of efforts targeted at developing distributed solutions. Early researchers [3, 21, 36, 41] pioneer in algorithm implementations on top of bulk synchronous systems, like Hadoop [2] and Spark MLlib [4]. In particular, Jiangtao et al. [41] discuss how to automatically compute a proper batch size in distributed environments for mini-batch computation. Such synchronous implementations provide strict convergence guarantee, but suffer from stragglers due to the global pre-defined barriers. Existing works tackling this problem basically fall into three categories: migration and backup, fine-grained synchronous, and asynchronous.

• Migration and backup: A straightforward solution is to dynamically migrate data from busy workers to idle workers to rebalance workloads [6, 15, 16]. Further, to reduce the expensive data migration costs, Harlap et al. [18] propose to pre-replicate data on a group of selected workers, so as to quickly migrate computation tasks. In addition, Hadoop and Spark [2, 4] support speculative execution by running straggling workers redundantly and using the output from the first successful run. Clearly, the two backup policies require additional memory and compute resources. Chen et al. [29] design another backup implementation. By setting a threshold b, the system skips synchronizing the slowest (called "backup") b workers, i.e., dropping updates from them. Different from that, our FSP can fully utilize such updates since they can also significantly contribute to convergence based on our tests in Section 5.4.

• Fine-grained synchronous systems: Confining the barrier operation to a subset of workers can perform fine-grained synchronous computation, which naturally reduces the number of workers blocked by stragglers. Specifically, the general dataflow system, Naiad [26], achieves this by pre-notifying a receiver that all data it requires have been ready. Then a receiver can start local computation without global barriers. Kadav et al. [20] further design an acknowledgement mechanism to avoid write conflicts on data kept at the receiver side, especially when the sender broadcasts data quickly. However, the fine-grained barrier is still pre-defined, which inevitably incurs waiting overheads for workers in the same subset.

• Asynchronous systems: Several works [5, 10, 13, 30, 42] aim at asynchronous computation without any barrier, such as TensorFlow [5] and Hogwild [30]. Although removing pre-defined barriers can solve the straggler problem, these systems potentially lose the convergence guarantee [19]. Recently, some researchers have attempted to hit a "sweet spot" between synchronous and asynchronous computation via a partially asynchronous mechanism. Altinigneli et al. [7] explore to run EM using GPU. Each processing unit can asynchronously update locally cached parameters until they converge. Subsequently, a global barrier is run to exchange intermediate updates among units. Similarly, Ho et al. [12, 19, 35] design a stale synchronous parallel framework where workers are allowed to use stale parameters got from the coordinator, but the staleness is user-controlled. However, specifying the staleness bound empirically is difficult. Worse, the objective function value may not change monotonously during the training process since workers (units) use inconsistent parameters to some extent. Besides, they still suffer from stragglers when updates are globally exchanged or the staleness bound is exceeded.

To summarize, different from works mentioned above, our flexible barrier mechanism provides the strict convergence guarantee as synchronous systems, while solving the straggler problem without any additional resource requirement.

7 CONCLUSION

This paper investigates the problem of EM computation. We propose a new flexible synchronous parallel framework that enables a coordinator to actively synchronize workers when necessary. In this way, fast workers can perform more useful computations, instead of blocking themselves to wait for stragglers, which speeds up convergence. A built-in adjustor dynamically collects execution statistics and then automatically computes a proper synchronous interval to gain optimal performance. Extensive experiments show that our proposals consistently outperform the state-of-the-art solutions. In future work, we plan to investigate whether the flexible barrier mechanism can be used in iterative graph algorithms.

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China (61433008, 61528203, and 61472071) and the U.S. NSF grant CNS-1217284. Zhigang Wang was a visiting student at UMass Amherst, supported by China Scholarship Council, when this work was performed. Authors also would like to thank anonymous reviewers and the shepherd, Asim Kadav, for their constructive comments.

REFERENCES

[1] 2017. Apache Giraph. (2017). http://giraph.apache.org/.

[2] 2017. Apache Hadoop. (2017). http://hadoop.apache.org/.

[3] 2017. Apache Mahout. (2017). http://mahout.apache.org/.

[4] 2017. Apache Spark. (2017). http://spark.apache.org/.

[5] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A system for large-scale machine learning. In *Proc. of OSDI*. 265–283.

[6] Umut Acar, Arthur Charguéraud, and Mike Rainey. 2013. Scheduling Parallel Programs by Work Stealing with Private Deques. In *Proc. of PPoPP*. ACM, 219–228.

[7] Muzaffer Can Altinigneli, Claudia Plant, and Christian Böhm. 2013. Massively parallel expectation maximization using graphics processing units. In *Proc. of KDD*. ACM, 838–846.

[8] Narayanaswamy Balakrishnan and Suvra Pal. 2016. Expectation maximization-based likelihood inference for flexible cure rate models with Weibull lifetimes. *Statistical methods in medical research* 25, 4 (2016), 1535–1563.

[9] M Emre Celebi, Hassan A Kingravi, and Patricio A Vela. 2013. A comparative study of efficient initialization methods for the k-means clustering algorithm. *Expert Systems with Applications* 40, 1 (2013), 200–210.

[10] Trishul M Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project Adam: Building an Efficient and Scalable Deep Learning Training System.. In *Proc. of OSDI*, Vol. 14. 571–582.

[11] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Machine learning* 20, 3 (1995), 273–297.

[12] Wei Dai, Abhimanu Kumar, Jinliang Wei, Qirong Ho, Garth Gibson, and Eric P Xing. 2015. High-Performance Distributed ML at Scale through Parameter Server Consistency Models. In *Proc. Of AAAI*.

[13] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. 2012. Large scale distributed deep networks. In *Advances in neural information processing systems*. 1223–1231.

[14] Arthur P Dempster, Nan M Laird, and Donald B Rubin. 1977. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the royal statistical society. Series B (methodological)* (1977), 1–38.

[15] James Dinan, D Brian Larkins, Ponnuswamy Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. 2009. Scalable work stealing. In *Proc. of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 53.

[16] James Dinan, Stephen Olivier, Gerald Sabin, Jan Prins, P Sadayappan, and Chau-Wen Tseng. 2007. Dynamic load balancing of unbalanced computations using message passing. In *Proc. of IPDPS*. IEEE, 1–8.

[17] J. C. Dunn. 1973. A fuzzy relative of the ISODATA Process and Its Use in Detecting Compact Well-Separated Clusters. *Journal of Cybernetics* 3 (1973), 32–57.

[18] Aaron Harlap, Henggang Cui, Wei Dai, Jinliang Wei, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, and Eric P Xing. 2016. Addressing the straggler problem for iterative convergent parallel ML.. In *Proc. of SoCC*. ACM, 98–111.

[19] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. 2013. More effective distributed ml via a stale synchronous parallel parameter server. In *Proc. of NIPS*. 1223–1231.

[20] Asim Kadav and Erik Kruus. 2016. ASAP: Asynchronous Approximate Data-Parallel Computation. *arXiv preprint arXiv:1612.08608* (2016).

[21] Wojtek Kowalczyk, Nikos A Vlassis, et al. 2004. Newscast EM.. In *Proc. of NIPS*. 713–720.

[22] Stuart Lloyd. 1982. Least squares quantization in PCM. *IEEE transactions on information theory* 28, 2 (1982), 129–137.

[23] James MacQueen et al. 1967. Some methods for classification and analysis of multivariate observations. In *Proc. of the fifth Berkeley symposium on mathematical statistics and probability*, Vol. 1. 281–297.

[24] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proc. of SIGMOD*. 135–146.

[25] Xiao-Li Meng and David Van Dyk. 1997. The EM Algorithm–an Old Folk-song Sung to a Fast New Tune. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 59, 3 (1997), 511–567.

[26] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proc. of SOSP*. ACM, 439–455.

[27] Radford M Neal and Geoffrey E Hinton. 1998. A view of the EM algorithm that justifies incremental, sparse, and other variants. In *Learning in graphical models*. Springer, 355–368.

[28] Thanh Minh Nguyen and QM Jonathan Wu. 2013. Fast and robust spatially constrained gaussian mixture model for image segmentation. *IEEE transactions on circuits and systems for video technology* 23, 4 (2013), 621–635.

[29] Xinghao Pan, Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. 2017. Revisiting Distributed Synchronous SGD. *arXiv preprint arXiv:1702.05800* (2017).

[30] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Proc. Of NIPS*. 693–701.

[31] David Sculley. 2010. Web-scale k-means clustering. In *Proc. of WWW*. ACM, 1177–1178.

[32] Supreeth Subramanya, Tian Guo, Prateek Sharma, David Irwin, and Prashant Shenoy. 2015. Spoton: A batch computing service for the spot market. In *Proc. of SoCC*. ACM, 329–341.

[33] Bo Thiesson, Christopher Meek, and David Heckerman. 2001. Accelerating EM for large databases. *Machine Learning* 45, 3 (2001), 279–299.

[34] Zhigang Wang, Yu Gu, Yubin Bao, Ge Yu, and Jeffrey Xu Yu. 2016. Hybrid Pulling/Pushing for I/O-Efficient Distributed and Iterative Graph Computing. In *Proc. of SIGMOD*. ACM, 479–494.

[35] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, and Eric P Xing. 2015. Managed communication and consistency for fast data-parallel iterative analytics. In *Proc. of SoCC*. ACM, 381–394.

[36] Jason Wolfe, Aria Haghighi, and Dan Klein. 2008. Fully distributed EM for very large datasets. In *Proc. of ICML*. ACM, 1184–1191.

[37] Xindong Wu, Vipin Kumar, J Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J McLachlan, Angus Ng, Bing Liu, S Yu Philip, et al. 2008. Top 10 algorithms in data mining. *Knowledge and information systems* 14, 1 (2008), 1–37.

[38] Yu-Wei Wu, Blake A Simmons, and Steven W Singer. 2015. MaxBin 2.0: an automated binning algorithm to recover genomes from multiple metagenomic datasets. *Bioinformatics* 32, 4 (2015), 605–607.

[39] Jiangtao Yin and Lixin Gao. 2016. Asynchronous distributed incremental computation on evolving graphs. In *Proc. of ECML/PKDD*. Springer, 722–738.

[40] Jiangtao Yin, Lixin Gao, and Zhongfei Mark Zhang. 2014. Scalable nonnegative matrix factorization with block-wise updates. In *Proc. of ECML/PKDD*. Springer, 337–352.

[41] Jiangtao Yin, Yanfeng Zhang, and Lixin Gao. 2012. Accelerating expectation-maximization algorithms with frequent updates. In *Cluster Computing, 2012 IEEE International Conference on*. IEEE, 275–283.

[42] Hyokun Yun, Hsiang-Fu Yu, Cho-Jui Hsieh, SVN Vishwanathan, and Inderjit Dhillon. 2014. NOMAD: Non-locking, stOchastic Multi-machine algorithm for Asynchronous and Decentralized matrix completion. *PVLDB* 7, 11 (2014), 975–986.