2012 IEEE International Conference on Cluster Computing

Accelerating Expectation-Maximization Algorithms with Frequent Updates

Jiangtao Yin*, Yanfeng Zhang†*, Lixin Gao*
*University of Massachusetts Amherst, USA
†Northeastern University, China
{jyin, yanfengzhang, lgao}@ecs.umass.edu

***Abstract*—Expectation Maximization is a popular approach for parameter estimation in many applications such as image understanding, document classification, or genome data analysis. Despite the popularity of EM algorithms, it is challenging to efficiently implement these algorithms in a distributed environment. In particular, many EM algorithms that frequently update the parameters have been shown to be much more efficient than their concurrent counterparts. Accordingly, we propose two approaches to parallelize such EM algorithms in a distributed environment so as to scale to massive data sets. We prove that both approaches maintain the convergence properties of the EM algorithms. Based on the approaches, we design and implement a distributed framework, FreEM, to support the implementation of frequent updates for the EM algorithms. We show its efficiency through three well-known EM applications: k-means clustering, fuzzy c-means clustering and parameter estimation for the Gaussian Mixture model. We evaluate our framework on both a local cluster of machines and the Amazon EC2 cloud. Our evaluation shows that the EM algorithms with frequent updates implemented on FreEM can run much faster than those implementations with traditional concurrent updates.**

I. Introduction

Discovering knowledge from a large collection of data sets is one of the most fundamental problems in many applications such as image understanding, document classification, or genome data analysis. Expectation-Maximization (EM) [6] is one of the most popular approaches in these applications. It estimates parameters for hidden variables by maximizing the likelihood. EM is an iterative approach that alternates between performing an Expectation step (E-step), which computes the distribution for the hidden variables using the current estimates for the parameters, and a Maximization step (M-step), which re-estimates parameters to be those maximizing the likelihood found in the E-step.

Due to its popularity, many methods for accelerating EM algorithms have been proposed. Some of them [11], [12] show that a partial E-step may accelerate convergence. Such a partial E-step selects only a subset of data points for computing the distribution. The advantage of the partial E-step is that it allows the M-step to be performed more frequently, so that the algorithm can leverage more up-to-date parameters to process data points and potentially accelerates convergence. Intuitively, updating the parameters frequently might incur additional overhead. However, the parameters typically depend on statistics of data sets that can be computed incrementally. That is, the cost of computing statistics grows linearly with the number of data points whose statistics have been changed in the E-step. As a result, performing frequent updates on the parameters does not necessarily introduce additional cost. We refer to the EM algorithm that updates the parameters frequently as *the EM algorithm with frequent updates*. In contrast, the traditional EM algorithm, which computes the distribution for all data points and then updates the parameters, is referred to as *the EM algorithm with concurrent updates*.

Despite the fact that the EM algorithm with frequent updates has the potential to speedup convergence, parallelizing it can be challenging. Although computing the distribution and updating statistics can be performed concurrently, parameters such as centroids of clusters are global parameters. Updating these global parameters has to be performed in a centralized location and all workers have to be synchronized. Synchronization in a distributed environment may incur considerable overhead. Therefore, we have to control the frequency of parameter update to obtain a good performance.

In this paper, we propose two approaches to parallelize the EM algorithm with frequent updates in a distributed environment: partial concurrent and subrange concurrent. In the *partial concurrent* approach, each E-step processes only a block of data points. The size of a block controls the frequency of parameter update. In the *subrange concurrent* approach, each E-step computes the distribution in a subrange of hidden variables instead of the whole range. The subrange size can determine the frequency of parameter update. We prove that both approaches maintain the convergence properties of the EM algorithms. We control the parameter update frequency by setting the block/subrange size, and provide strategies to determine the optimal values. Additionally, both approaches can scale to any number of workers/processors.

We design and implement a distributed framework, FreEM, for implementing the EM algorithm with frequent updates based on the two proposed approaches. FreEM eases the process of programming EM algorithms in a distributed environment. Programmers only need to specify the E-step and the M-step. The detailed mechanisms, such as data distribution, communication among workers, and frequency of M-step, are all handled automatically. As a result, it facilitates the process of implementing EM algorithms and accelerates the algorithms through frequent updates. We

978-0-7695-4807-4/12 $26.00 © 2012 IEEE
DOI 10.1109/CLUSTER.2012.81

evaluate FreEM in the context of three well-known EM applications, k-means clustering, fuzzy c-means clustering and parameter estimation for the Gaussian Mixture model. Our results show that the EM algorithm with frequent updates can run much faster than that with traditional concurrent updates. In addition, FreEM is more efficient than Hadoop [2], an open source implementation of MapReduce [5], in supporting the EM algorithms.

The rest of this paper is organized as follows. Section II describes the EM algorithm with frequent updates. Section III shows the advantages of frequent updates through three EM applications. Section IV presents our approaches to parallelize the EM algorithm with frequent updates. In Section V, we present the design, implementation and API of FreEM. Section VI is devoted to the evaluation results. Finally, we discuss related work in Section VII and conclude the paper in Section VIII.

II. EM ALGORITHMS

In a statistical model, suppose that we have observed the value of one random variable, X, which results from a parameterized family, $P(X|\theta)$. The value of another variable, Z, is hidden. Based on the observed data, we wish to find θ such that $P(X|\theta)$ is the maximum. In order to estimate θ, it is typical to introduce the log likelihood function: $L(\theta) = \log P(X|\theta)$. Suppose the data consists of n independent data points $\{x_1, ..., x_n\}$, and thereby the hidden variable can be decomposed as $\{Z_1, Z_2, ..., Z_n\}$. Then, $L(\theta) = \sum_{i=1}^{n} \log P(x_i|\theta)$. We assume that Z has a finite range for simplicity, but the result can be generalized. Thus, the probability $P(x_i|\theta)$ can be written in terms of possible value (z_i) of the hidden variable Z_i as: $P(x_i|\theta) = \sum_{z_i} P(x_i, z_i|\theta)$. When it is hard to maximize $L(\theta)$ directly, the EM algorithm is used to maximize $L(\theta)$ iteratively.

The EM algorithm leverages an iterative process to maximize $L(\theta)$. Each iteration consists of an E-step and a M-step. The E-step estimates the distribution of hidden variables, given the data points and the current estimates of the parameters. The M-step updates the parameters to be those maximizing the likelihood found in the E-step.

A. The EM Algorithm with Concurrent Updates

The EM algorithm with concurrent updates computes the distribution for all data points in its E-step. Formally, let Q_i be some distribution over z_i ($\sum_{z_i} Q_i(z_i) = 1$, $Q_i(z_i) \geq 0$). Such an EM algorithm starts with some initial guess at the parameters $\theta^{(0)}$, and then seeks to maximize $L(\theta)$ by iteratively applying the following two steps:

E-step: For each $x_i \in X$, set $Q_i(z_i) = P(z_i|x_i, \theta^{(t-1)})$.

M-step: Set $\theta^{(t)}$ to be the θ that maximizes $\sum_{i=1}^{n} E_{Q_i}[\log P(x_i, z_i|\theta)]$.

Here, the expectation E_{Q_i} is taken with respect to the distribution $Q_i(\cdot)$ over the range of Z in the E-step.

B. The EM Algorithm with Frequent updates

The EM algorithm with frequent updates attempts to accelerate the convergence by frequently updating the parameters. The intuition behind it is that the algorithm can leverage more up-to-date parameters to process data points and to potentially speedup convergence. In the EM algorithm, the distribution influences the likelihood of the parameters via some sufficient statistics. If the statistics can be updated incrementally, the cost of computing statistics will grow linearly with the number of data points whose statistics have been changed in the E-step. Therefore, performing frequent updates on the parameters does not necessarily introduce additional cost of computing statistics. However, it will incur extra overhead of deriving the parameters from the statistics. If the overhead is large, it is reasonable to compute the distribution for a subset of data points (or compute the distribution in a subrange of the hidden variable) and then update the parameters.

Updating the parameters frequently in the EM algorithm can be achieved by two approaches. One is *update by block*, which processes a block of data points in the E-step and then updates the parameters immediately. Its E-step can utilize the up-to-date parameters to process another block of data points. Obviously, when selecting the whole set of data points as a block, the EM algorithm with update by block is actually the EM algorithm with concurrent updates. Its two steps can be described as following:

E-step: Pick a block of data points, B_m ($B_m \subseteq X$), and for each $x_i \in B_m$,

Set $Q_i^{(t)}(z_i) = P(z_i|x_i, \theta^{(t-1)})$.

M-step: Set $\theta^{(t)}$ to be the θ that maximizes $\sum_{i=1}^{n} E_{Q_i}[\log P(x_i, z_i|\theta)]$.

The other one is *update by subrange*, which recomputes the distribution over a subrange of the hidden variable and then updates the parameters. Its E-step can leverage the up-to-date parameters to recompute the distribution over another subrange. The EM algorithm with update by subrange starts with some initial guess at the parameters $\theta^{(0)}$ and some guess at the distribution $Q_i^{(0)}$, and then seeks to maximize $L(\theta)$ by iteratively applying the following two steps:

E-step: Select a subrange of Z, R_{sub}, for each $x_i \in X$,

Let $C_{R_{sub}} = \sum_{z_i \in R_{sub}} Q_i^{(t-1)}(z_i)$;

Set $Q_i^{(t)}(z_i) = P(z_i|x_i, \theta^{(t-1)}) * C_{R_{sub}}$.

M-step: Set $\theta^{(t)}$ to be the θ that maximizes $\sum_{i=1}^{n} E_{Q_i}[\log P(x_i, z_i|\theta)]$.

We can also combine the two approaches to achieve updating the parameters frequently. Such a combined version selects a subrange of Z and computes the distribution for a block of data points under the subrange in its E-step, and then performs the M-step to update the parameters. Obviously, either approach is a special case of the combined version. Moreover, even the combined version maintains the convergence properties of the EM algorithm. A brief proof

that the EM algorithm with frequent updates (the combined version) converges is provided in Appendix, and a complete proof is offered in [14].

Updating the parameters in this paper means only the operations of deriving the parameters from the statistics (not including the operations of updating the statistics). In Section III, we will illustrate what such statistics are and show the advantages of frequent updates via EM applications.

The EM algorithm with frequent updates brings the intuition that updating the parameters the more frequently the better. However, updating the parameters needs to be done in a centralized way when the algorithm is performed in a distributed environment. Synchronizing the global resources may result in significant overhead. In Section IV, we will discuss how to parallelize the EM algorithm with frequent updates, and how to control the frequency of parameter update to achieve a good performance.

III. Applications of the EM Algorithm

In this section, we describe three problems which the EM algorithm can be applied to, k-means clustering, Fuzzy c-means clustering and parameter estimation for the Gaussian Mixture model. We illustrate how to incrementally compute the statistics and how to derive the parameters from the statistics when applying the EM algorithm to these problems. By introducing the statistics, the operations of computing the parameters are divided into the operations of incrementally updating the statistics and the operations of deriving the parameters from the statistics. The cost of updating the statistics through a pass of all data points is fixed, no matter how frequently the algorithm updates the parameters. The frequent updates increase only the cost of deriving the parameters from the statistics. The more frequently it updates the parameter, the more cost the algorithm will incur. Also, we show the advantages of performing frequent updates.

A. K-means

K-means clustering [9] aims to partition n data points $\{x_1, x_2, ..., x_n\}$ into k ($k \leq n$) clusters $\{c_1, c_2, ..., c_k\}$ so as to minimize the objective function: $\langle f \rangle = \sum_{i=1}^{k} \sum_{x_j \in c_i} \parallel x_j - \mu_{c_i} \parallel^2$, where $\mu_{c_i} = \frac{1}{|c_i|} \sum_{x_j \in c_i} x_j$ is the centroid of cluster c_i.

The most common algorithm of k-means clustering (Lloyd's algorithm [8]) can be considered as an application of the EM algorithm. Its E-step assigns points to the cluster with the closest mean. Its M-step updates the centroids (parameters) for all clusters. Let S_i ($S_i = \sum_{x_j \in c_i} x_j$) and W_i ($W_i = |c_i|$) be the statistics. The centroid of one cluster (*e.g.*, i) can be easily obtained by $\mu_i = \frac{S_i}{W_i}$. If a particular point x_i changes its cluster assignment from c to c', the statistics can be incrementally updated as follows:

$$S_c = S_c - x_i, \quad S_{c'} = S_{c'} + x_i;$$
$$W_c = W_c - 1, \quad W_{c'} = W_{c'} + 1.$$

B. Fuzzy C-means

Given a set of data points $\{x_1, x_2, ..., x_n\}$, Fuzzy c-means (FCM) [4] aims to assign n data points into C clusters $\{c_1, c_2, ..., c_k\}$ so as to minimize the objective function: $J_m = \sum_{i=1}^{n} \sum_{j}^{C} \mu_{ij}^m \parallel x_i - c_j \parallel^2$, where m ($m > 1$) is the fuzzy factor, $\mu_{ij} = \frac{1}{\sum_{k=1}^{C} (\frac{\|x_i - c_j\|}{\|x_i - c_k\|})^{\frac{2}{m-1}}}$ is the degree of membership of x_i belonging to cluster j, and $c_j = \frac{\sum_{i=1}^{N} \mu_{ij}^m x_i}{\sum_{i=1}^{N} \mu_{ij}^m}$ is the centroid of cluster j.

If we describe FCM in the EM setting, its E-step updates the degree of membership for all points, and its M-step updates the centroids (parameters) for all clusters. Let W_j ($W_j = \sum_{i=1}^{n} \mu_{ij}^m$) and X_j ($X_j = \sum_{i=1}^{n} \mu_{ij}^m x_i$) be the statistics in FCM. The centroid of one cluster (*e.g.*, j) can be easily obtained by $c_j = \frac{X_j}{W_j}$. For a data point x_i, if its degree of membership to cluster j changes from μ_{ij} to μ'_{ij}, the statistics can be incrementally updated as follows:

$$W_j = W_j - (\mu_{ij})^m + (\mu'_{ij})^m,$$
$$X_j = X_j + ((\mu'_{ij})^m - (\mu_{ij})^m) x_i.$$

C. Gaussian Mixture Model

Given a set of data points $\{x_1, x_2, ..., x_n\}$ which are generated by a mixture of k Gaussians, parameter estimation for the Gaussian Mixture model (GMM) aims to find the means and covariances of the k Gaussians and the weights that specify how likely each Gaussian is to be chosen so as to maximize the objective function: $\ell = \frac{1}{n} \sum_{i=1}^{n} \log(\sum_{j=1}^{k} \omega_j \phi(x_i | \mu_j, \Sigma_j))$, where $\phi(x_i | \mu_j, \Sigma_j) = \frac{1}{\sqrt{(2\pi)^d \cdot |\Sigma_j|}} \cdot e^{-\frac{1}{2}(x_i - \mu_j)^T \cdot \Sigma_j^{-1} \cdot (x_i - \mu_j)}$ represents the probability of a point coming from a Gaussian source. The parameters (weight, mean and covariance) of a Gaussian (*e.g.*, j) are computed by the following equations:

$$\omega_j = \frac{1}{n} \sum_{i=1}^{n} \gamma_{ij}, \qquad \mu_j = \frac{\sum_{i=1}^{n} \gamma_{ij} x_i}{\sum_{i=1}^{n} \gamma_{ij}},$$
$$\Sigma_j = \frac{\sum_{i=1}^{n} \gamma_{ij} (x_i - \mu_j)(x_i - \mu_j)^T}{\sum_{i=1}^{n} \gamma_{ij}},$$

where γ_{ij} represents the probability of a point coming from a Gaussian, which is given by: $\gamma_{ij} = \frac{\omega_j \phi(x_i | \mu_j, \Sigma_j)}{\sum_{j=1}^{k} \omega_j \phi(x_i | \mu_j, \Sigma_j)}$.

When describing GMM in the EM setting, its E-step estimates the probability of a point from a Gaussian for all points, and its M-step updates the parameters of Gaussians.

The covariance matrix Σ is typically assumed to be diagonal to facilitate the computation of its inverse and determinant. Under such assumption, the statistics in the GMM algorithm are as follows:

$$R_j = \sum_{i=1}^{n} \gamma_{ij}, X_j = \sum_{i=1}^{n} \gamma_{ij} x_i, S_j = \sum_{i=1}^{n} \gamma_{ij} x_i^2.$$

Given the statistics, the parameters $\omega_j = R_j / n$, $\mu_j = X_j / R_j$ and $\Sigma_j = S_j / R_j - X_j^2 / R_j^2$ can be easily obtained.

For a point x_i, if its probability to the source j changes from γ'_{ij} to γ_{ij}, the statistics can be computed as follows:

$$\begin{aligned} R_j =& R_j + \gamma_{ij} - \gamma'_{ij}, \\ X_j =& X_j + (\gamma_{ij} - \gamma'_{ij})x_i, \\ S_j =& S_j + (\gamma_{ij} - \gamma'_{ij})x_i^2. \end{aligned}$$

D. Advantages of Performing Frequent Updates

Since the EM algorithm with frequent updates utilizes the up-to-date parameters to estimate the distribution, it intuitively outperforms its concurrent update counterpart. In a preliminary study, we performed multiple experiments on a single machine to demonstrate the advantages of frequent updates. Our experimental results show the EM algorithm with frequent updates converges faster compared to that with concurrent updates. The results also exhibit that the frequency of parameter update significantly affects the performance. The detailed results can be found in [14]. Moreover, some previous results [11], [12] also showed the advantages of the frequent updates for EM algorithm in a single machine setting. However, the EM algorithm with frequent updates in a single machine does not scale. Parallelizing the EM algorithm with frequent updates is important for real-world applications. The rest of this paper will focus on parallelizing the EM algorithm with frequent updates.

IV. Parallelizing Frequent Updates

The previous sections show that the EM algorithm with frequent updates is more efficient than that with concurrent updates. However, parallelizing frequent updates is challenging. Although computing the distribution and incrementally updating the local statistics in the E-step can be performed concurrently in each worker, updating the parameters in the M-step, which is based on the global statistics, needs to be done in a centralized way. When processing the distributed points, the algorithm has to synchronize the global statistics frequently. Synchronizing the global resources in a distributed environment may result in considerable overhead. Therefore, we need to control the parameter update frequency to achieve a good performance. In this section, we first illustrates a natural method to parallelize the EM algorithm with concurrent updates. Then, we present two methods to parallelize the EM algorithm with frequent updates. Both of them can control the parameter update frequency. Moreover, in all the parallel methods, the input data is randomly divided into multiple equal size partitions, and each worker holds one partition. The data is kept in the same worker throughout the iterative process to avoid the expensive data shuffling among workers.

A. Concurrent Method

In this subsection, we illustrate a traditional method to parallelize concurrent updates. In this method, each worker computes the distribution for its local data points and updates the local statistics concurrently based on the parameters. After each worker finishes processing its local data points, all of them synchronize to derive the parameters from the global statistics. Then, each worker utilizes the updated parameters to compute the distribution in the next iteration. We refer to the method as *concurrent* method.

B. Partial Concurrent Method

Our first method to parallelize the EM algorithm with frequent updates is a parallel version of the update by block approach in Section II. Recall that the update by block approach selects a block of data points for computing the distribution and then updates the parameters. The block size can control the parameter update frequency. As shown in Figure 1, our first parallel method allows each worker to pick a block of its local data points for computing the distribution and updating the local statistics. After processing the data points in the picked blocks, all the workers synchronize to derive the parameters from the global statistics. Then each worker leverages the updated parameters to compute the distribution for another block. All the blocks are of the same size m. Each worker rotates the block on its local data points. Since the data points in the picked blocks can be processed concurrently, we refer to this method as *partial concurrent* method. Obviously, the concurrent method is an extreme case of the partial concurrent method (when each worker selects all its local data points as one block).

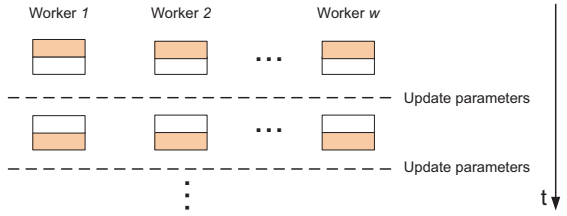


Figure 1. Process of the partial concurrent method. The colored box indicates the picked block of data points for computing the distribution.

The size of the block (*i.e.*, m) plays an important role on the efficiency of the partial concurrent method. It indicates the trade-off between the gain from computing the distribution with the frequently updated parameters and the cost from updating the parameters. Setting the size too small may incur considerable overhead for updating the parameters. Setting the size too large may degrade the effect of the frequent updates. Nevertheless, a quite large range of the block size can improve the performance. The optimal block size will be discussed in Section V-D. Our framework also provides a recommended block size.

C. Subrange Concurrent Method

Our second method to parallelize the EM algorithm with frequent updates corresponds to the update by subrange approach in Section II. Recall that the update by subrange approach recomputes the distribution over the subrange of

hidden variables. As shown in Figure 2, our second parallel method allows each worker to recompute the distribution among the subrange for its local data points and to update its local statistics. After each worker finishes recomputing the distribution among the subrange for all of its local data points, all the workers synchronize to compute the parameters based on the global statistics. Then, each worker utilizes the updated parameters to recompute the distribution under another subrange in next iteration. Since all the data points can be processed concurrently under the subrange, we refer to the second method as *subrange concurrent* method. The subrange is randomly picked from the whole range of hidden variables. The concurrent method is an extreme case of the subrange concurrent method as well (when the whole range is picked as the subrange).

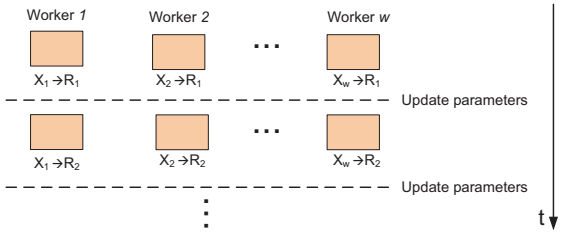


Figure 2. Process of the subrange concurrent method. Each worker recomputes the distribution among the subrange (R_i) for all of its local data points (X_j).

The subrange concurrent method might be more suitable for a "winner-take-all" version of EM application (*e.g.*, k-means), which constrains that one single value of the hidden variable is assigned probability 1 and all other values are assigned probability 0 (in k-means, a data point belongs to its current cluster in probability 1 and belongs to all other clusters in probability 0). In such an application, if a subrange does not include the value of probability 1, it is not necessary to recompute the distribution among the subrange. By avoiding unnecessary computation, a worker may dramatically reduce the time of processing data points in one iteration. Within the running time of one iteration of the concurrent method, the subrange concurrent method may proceed many iterations. Therefore, although the subrange concurrent method may increase (or decrease, here we assume "increase") the objective function less than the concurrent method in one single iteration, it still may increase the objective function faster. Moreover, the distribution for most of data points usually will not change after first several iterations under the concurrent method, and thus the objective function probably increases slowly after first several iterations. Consequently, the concurrent method probably does not increase the objective function much more than the subrange concurrent method in one single iteration, which makes the subrange concurrent method more superior.

Like the block size in the partial concurrent method, the size of the subrange also impacts the efficiency of the subrange concurrent method. We will also discuss the optimal subrange size in Section V-D.

V. FreEM

In this section, we propose FreEM, a distributed framework for efficient implementation of an EM algorithm. All the parallel methods mentioned in the previous section, including concurrent, partial concurrent, and subrange concurrent, are supported by our framework. FreEM is built on top of an in-memory version of iMapReduce [15]. The in-memory version of iMapReduce supports iterative process and loads data into memory for efficient data access. FreEM also provides high-level APIs, which are exposed to users for easily implementing EM algorithms.

A. Design of the Framework

Our framework consists of a number of *basic workers* and an *enhanced worker*. Each basic worker essentially leverages user-defined functions to compute the distribution and to update the parameters. Besides these operations, the enhanced worker also picks the subrange of hidden variable for all the workers under the subrange concurrent method. Each worker stores a partition of the data points, the distribution of the corresponding hidden variables, the local statistics (the statistics for a worker's local data points) and the parameters in memory. The partition of points and the distribution are maintained in a key-value store, *point-based table*. Also, the local statistics and the parameters is maintained in a key-value store, *parameter-based table*.

B. Implementation of the Framework

Each worker in our framework has one pair of map and reduce tasks. In general, the map task performs the M-step, and the reduce task performs the E-step. The map task of the enhanced worker takes charge of picking the subrange of hidden variables. Both the point-based table and the parameter-based table of each worker is maintained by its reduce task.

To implement an EM algorithm, a user only needs to override several APIs. FreEM will automatically convert the EM algorithm to iMapReduce jobs. The first job is used to split the input data into multiple equal size partitions. The second job executes the EM algorithm, which consists of many iterations. In the first iteration, each map task utilizes a user-defined function (API 1) to obtain the initial guess of the parameters. Then, each map task sends the parameters to its paired reduce task. Each reduce task first loads one partition of the input data and then leverages a user-defined function (API 2) to compute the distribution and to initialize its local statistics. After that, a reduce task broadcasts its local statistics to all map tasks. In each of the following iterations, each map task uses a user-defined function (API 3) to accumulate the local statistics it received to the global statistics. When it receives the local statistics from all reduce tasks, a map task uses another user-defined function (API 4) to derive the parameters from the global statistics. Then, each map task sends the updated parameters to its paired

reduce task. A reduce task leverages another user-defined function (API 5) to recompute the distribution (under a given subrange) and to incrementally update its local statistics based on the updated parameters. After it finishes processing the given block of data points, a reduce task broadcasts its updated local statistics to all map tasks again. Such iterative process continues until the number of iterations exceeds a threshold, when our framework terminates all the tasks. Note that the map task of the enhanced worker also picks a subrange and broadcasts it to all reduce tasks under the subrange concurrent method.

C. API

FreEM provides several high-level APIs, which are exposed to users for easily implementing an EM algorithm. The APIs are as follows:

1. `void initPara(Para,Points)`: specify the initial guess at the parameters.
2. `void initLocalStat(Dist,LocalStat,Para, Points)`: compute the distribution based on the initial guess at the parameters, and initialize the local statistics.
3. `void accuStat(LocalStat,GlobalStat)`: accumulate the local statistics to the global statistics.
4. `void updatePara(GlobalStat,Para)`: update the parameters based on the global statistics.
5. `void Estep(Dist,Para,SubRange,LocalStat, Points)`: recompute the distribution under the given subrange based on current parameters, and incrementally update the local statistics.

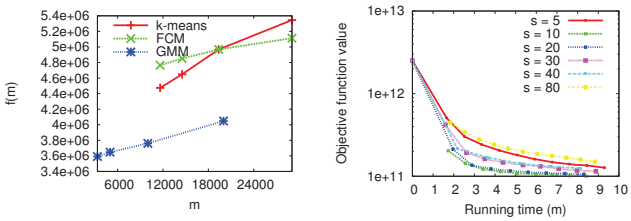
D. Setting Parameters for Parallel Methods

The size of the block in the partial concurrent method and the size of the subrange in the subrange concurrent method can significantly impact the performance of the algorithm. In this section, we discuss how to determine the optimal block size and how to seek the optimal subrange size.

1) Optimal block Size: For the partial concurrent method, let m be the block size. We use T_{sgl} to represent the average time of processing one data point, consisting of the time for computing the distribution and the time for updating local statistics, and use T_{vhd} to represent the time spending on updating the parameters, consisting of the time for accumulating the global statistics, the time for updating the parameters, and the time of synchronization. Let $F(m)$ be total number of data points that one worker needs to process in the E-step for reaching a specified objective function value (*i.e.*, convergence point) when the block size is m. Then, $\frac{F(m)}{m}$ is the total number of iterations. Thus, the total running time for reaching the convergence point is $\{F(m) \cdot T_{sgl} + \frac{F(m)}{m} \cdot T_{vhd}\}$. Therefore, the optimal m is given by:

$$\arg\min_{m}\{F(m) \cdot T_{sgl} + \frac{F(m)}{m} \cdot T_{vhd}\},$$

where T_{sgl} and T_{vhd} can be measured. The key of finding the optimal m is the function $F(m)$.



(a) Partial Concurrent (b) Subrange Concurrent

Figure 3. Deriving optimal size.

We estimate $F(m)$ for different applications of the EM algorithm on a local cluster. The result, as shown in Figure 3a, demonstrates that $F(m)$ is roughly a linear function of m, *i.e.*, $F(m) = a \cdot m + b$. Then, we can derive the optimal block size m^*:

$$m^* = \sqrt{\frac{b \cdot T_{vhd}}{a \cdot T_{sgl}}}.$$

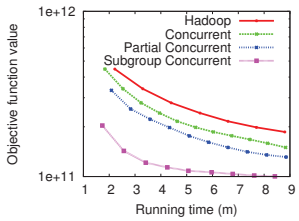
Among the factors determining the optimal block size, only T_{vhd} and T_{sgl} can be easily measured. Therefore, we consider m is linear in $\sqrt{\frac{T_{vhd}}{T_{sgl}}}$. We can explore different settings of $m/\sqrt{\frac{T_{vhd}}{T_{sgl}}}$ to seek the optimal block size. In our framework, we set $m/\sqrt{\frac{T_{vhd}}{T_{sgl}}}$ to be 300 by default. The default setting achieves near optimal performance as will be shown in Section VI.

Our framework measures T_{vhd} and T_{sgl} in the following way. When it executes an EM algorithm, FreEM first sets the block size as a pre-defined number (*e.g.*, $\frac{n}{4 \cdot w}$, where n is the total number of data points and w is the number of workers). Then, each worker measures its own T_{vhd} and T_{sgl}, and reports their values to the enhanced worker in each iteration. The enhanced worker accumulates both of them, respectively. After a few (*e.g.*, 3) iterations, the enhanced worker computes the average values of both T_{vhd} and T_{sgl} and specifies $300 \cdot \sqrt{\frac{T_{vhd}}{T_{sgl}}}$ as the optimal block size.

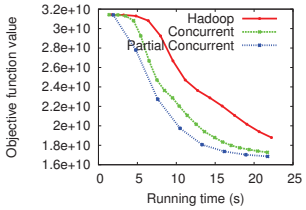
2) Optimal Subrange Size: For the subrange concurrent method, let s be the subrange size and r be the size of the whole range. Suppose $\Delta f(s)$ is the averaging increase of the objective function for computing the distribution among the subrange for all data points and updating the parameters when the subset size is s. Since the time of processing one data point is usually proportional to the subrange size, $\frac{s}{r} \cdot T_{sgl}$ is the time for processing one data point under the subrange concurrent method. Therefore, $\frac{n}{w} \cdot \frac{s}{r} \cdot T_{sgl} + T_{vhd}$ is the running time of processing all data points in one iteration. Consequently, the optimal subrange size is given by:

$$\arg\max_{s}\{\frac{\Delta f(s)}{\frac{n}{w} \cdot \frac{s}{r} \cdot T_{sgl} + T_{vhd}}\}.$$

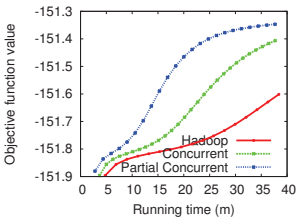
We use empirical approaches to seek the optimal subrange size. Our experimental results reveal that if one subrange size is better than another during the initial iterations, it



(a) K-means on Covtype

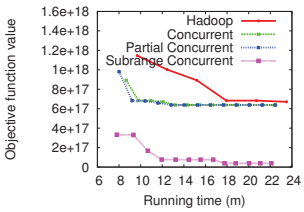


(b) FCM on Covtype

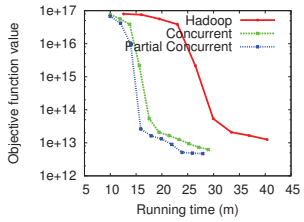


(c) GMM on Synth-M

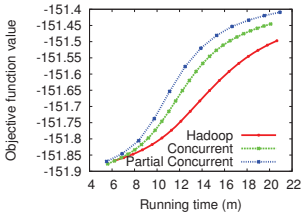
Figure 4. Convergence speed on the local cluster.



(a) K-means on KDDCUP



(b) FCM on KDDCUP



(c) GMM on Synth-L

Figure 5. Convergence speed on the Amazon EC2 cluster.

is also better in the following iterations (*e.g.*, as shown in Figure 3b for k-means). Given the observation, we can try several subrange sizes, and pick the one that achieves the best performance in the first several iterations.

Also, we provide a scheme to judge when the subrange concurrent method is superior to the concurrent method. Obviously, we can expect that the subrange concurrent method will outperform the concurrent method when the following inequality holds.

$$\frac{\Delta f(s)}{\frac{n}{w}\cdot\frac{s}{r}\cdot T_{sgl}+T_{vhd}} > \frac{\Delta f(r)}{\frac{n}{w}\cdot T_{sgl}+T_{vhd}}. \quad (1)$$

From Inequation 1, we can derive another inequation, which is as follows:

$$\frac{\Delta f(s)}{\Delta f(r)} > \frac{r}{s}+\frac{\frac{r-s}{r}\cdot T_{vhd}}{\frac{n}{w}\cdot T_{sgl}+T_{vhd}}. \quad (2)$$

All the factors in the right side of Inequation 2 either are known or can be measured. Accordingly, it provides a nice bound to estimate whether the subrange concurrent method achieves better performance than the concurrent method.

VI. Evaluation

In this section, we evaluate FreEM on the applications described in Section III. All the parallel methods, including concurrent, partial concurrent, and subrange concurrent, are implemented and evaluated. We also compare the concurrent method on FreEM with that on Hadoop, an open source implementation of MapReduce.

A. Experiment Setup

We build a local cluster and a large-scale cluster on Amazon EC2 [1]. The local cluster consists of 4 machines. The Amazon cluster consists of 40 medium instances.

Real-world data sets from UCI Machine Learning Repository [3] and synthetic data sets are leveraged to evaluate the applications. The synthetic data sets are generated in such a way: each dimension of one data point follows a Gaussian distribution with random mean and standard deviation 1.0. The data sets are summarized in Table I.

Table I
Data Sets Summary

Algorithm	Data set	# Points	Dim
k-means/FCM	Covtype	581,012	54
	KDDCUP	4,898,431	42
GMM	Synth-M	400,000	60
	Synth-L	1,000,000	60

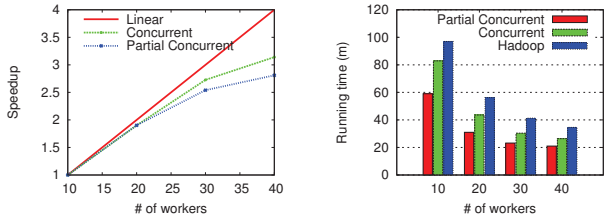
B. Convergence Speed

FreEM allows the EM algorithm to frequently update the parameters and leverage the up-to-date parameters in its E-step. Therefore, the EM algorithm with frequent updates has the potential to reach the convergence point with less workload, compared to that with concurrent updates. To evaluate the effect of frequent updates, we compare the convergence speed of the partial/subrange concurrent method with that of the concurrent method. In addition, since MapReduce is a popular framework, we utilize the convergence speed of the concurrent method on Hadoop as the base line.

The convergence speed evaluation is performed on both the local cluster and the Amazon EC2 cluster. All the methods start with the same initial guess, when compared on the same data set. We set the number of clusters/sources as 80 for all experiments. Figure 4 and Figure 5 show the performance comparison. We can see that the partial concurrent method converges faster than the concurrent method on all the three EM applications. The subrange concurrent

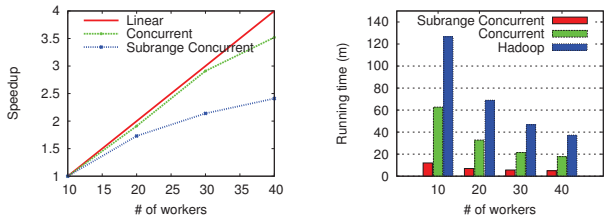
method converges faster and converges to a much better point than the concurrent method on k-means. Unfortunately, the subrange concurrent method seems to be slower than the concurrent method on FCM and GMM with several subrange sizes we test. Additionally, the convergence speed of the concurrent method on FreEM is much faster than that on Hadoop. The reasons are twofold. One is that our framework maintains data in memory and thus avoids repeatedly loading data. The other is that FreEM is built on top of iMapReduce, which is more efficient in supporting iterative process than Hadoop by using persistent map and reduce tasks. For example, iMapReduce is more efficient than Hadoop in supporting graph based iterative algorithms [15], [16]. Additionally, according to the experimental results, it seems that the subrange concurrent method is suitable for "winner-take-all" version of EM applications and the partial concurrent method is suitable for the other ("soft") version of EM applications.

C. Scaling Performance



(a) Speedup of partial concurrent (b) Performance comparison

Figure 6. Scaling performance of the partial concurrent method.



(a) Speedup of subrange concurrent (b) Performance comparison

Figure 7. Scaling performance of the subrange concurrent method.

Figure 6 and Figure 7 plot the speedup of FreEM as the number of workers increases from 10 to 40. The speedup is measured relative to the running time of 10 workers. The speedup of the partial concurrent method is tested on GMM, and that of the subrange concurrent method is measured on k-means. The speedup of the concurrent method is also evaluated to be a reference point.

Figure 6 shows that both the concurrent method and the partial concurrent method exhibit good speedups. The concurrent method demonstrates a better speedup, since it updates the parameters only once through one pass of all data points and thus incurs less synchronization overhead. Note that the bases of computing speedups for both methods are different, and thus a better speedup does not necessarily mean a shorter running time. As shown in Figure 6b, the partial concurrent method still converges faster than the concurrent method even on 40 workers. Since it has a better speedup, the concurrent method will obtain the same convergence speed as the partial concurrent method when the number of workers reaches some point. At that point, the partial concurrent method will degrade to the concurrent method by setting the right block size. For similar reasons, the concurrent method also exhibits a better speedup than the subrange concurrent method. However, the subrange concurrent method still runs much faster than the concurrent method even on 40 workers, as shown in Figure 7b.

D. Verifying Optimal Block Size

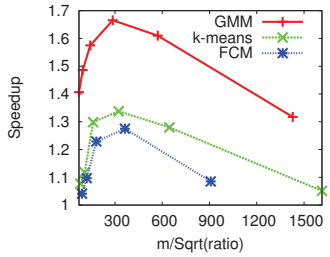


Figure 8. Speedup. X-axis represents the values of $m/\sqrt{\frac{T_{vhd}}{T_{sgl}}}$.

In Section V-D, we discussed the optimal block size depends on several factors. Since only T_{vhd} and T_{sgl} can be easily measured, we set the block size based on them. Our framework sets the block size m in proportional to $\sqrt{\frac{T_{vhd}}{T_{sgl}}}$. We perform experiments of all the three EM applications on our local cluster to see the effects of various settings of $m/\sqrt{\frac{T_{vhd}}{T_{sgl}}}$. Figure 8 shows the speedup with different settings. From the figure, we can see that all the applications demonstrate the best speedup when $m/\sqrt{\frac{T_{vhd}}{T_{sgl}}}$ is set to be around 300. This is the reason why our framework sets $m/\sqrt{\frac{T_{vhd}}{T_{sgl}}} = 300$ by default.

VII. Related Work

The EM algorithm has been applied very widely. Due to the popularity of the EM algorithm, many approaches for accelerating it have been proposed. For example, Dempster *et al.* [6] and Meng *et al.* [10] present a partial M-step may accelerate the algorithm when maximizing the likelihood in the M-step is inefficient. Such a partial M-step attempts to find the new estimates for the parameters improving the likelihood rather than maximizing it. In contrast, our work focuses on how to frequently perform the M-step to accelerate the algorithm. As the most relevant works, the works of Neal *et al.* [11] and Thiesson *et al.* [12] also show a partial E-step which selects a block of data points for computing the distribution may accelerate the EM algorithm in the single machine setting. Neal *et al.* [11] prove that such a variant of the EM algorithm converges. Thiesson *et*

al. [12] provide an empirical method to figure out the near optimal block size. Our proof is inspired by the work of Neal *et al.*, but goes further. Specifically, we prove that not only selecting a block of data points for computing the distribution but also computing the distribution under a subrange of hidden variables can guarantee the convergence. Compared to the work of Thiesson *et al.*, which is in the single machine setting, our work considers the scenario of a distributed environment. We propose a distributed framework for efficiently implementing the EM algorithm with frequent updates.

There are a number of efforts targeted on parallelizing the EM algorithm as well. Most of them focused on efficiently updating the parameters in the M-step. For examples, Wolfe *et al.* [13] propose an approach to distribute both the E-step and the M-step based on MapReduce. Kowalczyk *et al.* [7] present a gossip-based distributed implementation of the EM algorithm for GMM. While our work has a different focus: we study how to frequently update the parameters to speed up convergence for a wide class of EM algorithms.

VIII. Conclusion

Motivated by the observations that the EM algorithm performing the frequent updates is much more efficient than it performing the concurrent updates, we propose two approaches to parallelize the EM algorithm with frequent updates in a distributed environment so as to scale to massive data sets. Also, we prove that the EM algorithm with frequent updates converges. To support the efficient implementation of frequent updates for the EM algorithm, we design and implement a distributed framework, FreEM. We deploy FreEM on both a local cluster and the Amazon EC2 cloud, and evaluate its performance in the context of three well-known EM applications: k-means, FCM and GMM. The evaluation results show that the EM algorithm with frequent updates can run much faster than it with traditional concurrent updates even when both are implemented on FreEM. In addition, since FreEM is on top of iMapReduce which is more efficient than MapReduce in supporting iterative algorithms, FreEM is more efficient than MapReduce in supporting the EM algorithm.

Acknowledgments

The authors are grateful to the anonymous reviewers for their insightful comments and suggestions. This work is partially supported by NSF grant CCF-1018114. Yanfeng Zhang was a visiting student at UMass Amherst when this work was performed.

References

[1] Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/.
[2] Hadoop. http://hadoop.apache.org/.
[3] UCI Machine Learning Repository. http://archive.ics.uci.edu/ml/datasets.html.
[4] J. C. Bezdek. *Pattern Recognition with Fuzzy Objective Function Algorithms*. Kluwer Academic Publishers, 1981.
[5] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI '04*, 2004.
[6] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society, Series B*, 39(1), 1977.
[7] W. Kowalczyk and N. Vlassis. Newscast EM. In *NIPS '04*, pages 713–720, 2005.
[8] S. Lloyd. Least squares quantization in PCM. *Information Theory, IEEE Transactions on*, 28(2):129 – 137, mar 1982.
[9] J. B. Macqueen. Some methods of classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.
[10] X. L. Meng and D. B. Rubin. Maximum Likelihood Estimation via the ECM Algorithm: A General Framework. *Biometrika*, 80(2):267–278, 1993.
[11] R. Neal and G. E. Hinton. A view of the EM algorithm that justifies incremental, sparse, and other variants. In *Learning in Graphical Models*, pages 355–368, 1998.
[12] B. Thiesson, C. Meek, and D. Heckerman. Accelerating EM for large databases. *Mach. Learn.*, 45(3):279–299, Dec. 2001.
[13] J. Wolfe, A. Haghighi, and D. Klein. Fully distributed EM for very large datasets. ICML '08, 2008.
[14] J. Yin, Y. Zhang, and L. Gao. Accelerating expectation-maximization algorithms with frequent updates. Tech. Rep., 2012. http://rio.ecs.umass.edu/mnilpub/papers/freem.pdf.
[15] Y. Zhang, Q. Gao, L. Gao, and C. Wang. iMapReduce: A distributed computing framework for iterative computation. In *DataCloud '11*, pages 1112 –1121.
[16] Y. Zhang, Q. Gao, L. Gao, and C. Wang. PrIter: A distributed framework for prioritized iterative computations. In *SOCC '11*, pages 13:1–13:14, 2011.

Appendix

For proving the convergence of the EM algorithm with frequent updates, we first consider the following derivation:

$$L(\theta) = \sum_{i=1}^{n} \log P(x_i|\theta) = \sum_{i=1}^{n} \log \sum_{z_i} Q_i(z_i) \frac{P(x_i, z_i|\theta)}{Q_i(z_i)} \geq \sum_{i=1}^{n} \sum_{z_i} Q_i(z_i) \log \frac{P(x_i, z_i|\theta)}{Q_i(z_i)}.$$

The last step of this derivation is given by Jensen's inequality. When $Q_i(z_i) = P(z_i|x_i, \theta)$ for any i, the last step of the derivation holds with equality. Let

$$J(Q, \theta) = \sum_{i=1}^{n} \sum_{z_i} Q_i(z_i) \log \frac{P(x_i, z_i|\theta)}{Q_i(z_i)},$$

then we have $L(\theta) \geq J(Q, \theta)$. We assume that $P(x_i, z_i|\theta)$ is a continuous function of θ. We can show that if the local maximum of $J(Q, \theta)$ occurs at Q^* and θ^*, the local maximum of $L(\theta)$ occurs at θ^* as well. Hence, if a variant of the EM algorithm gradually increases $J(Q, \theta)$, it will converge to a local maximum of $L(\theta)$. Additionally, we can prove that each iteration of the EM algorithm with frequent updates either improves $J(Q, \theta)$ or leaves it unchanged. Therefore, we have the following theorem.

Theorem A.1: The EM algorithm with frequent updates converges to a local maximum of $L(\theta)$.