J. Parallel Distrib. Comput. 111 (2018) 65–75

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Accelerating distributed Expectation–Maximization algorithms with frequent updates[☆]

Jiangtao Yin[a,*], Yanfeng Zhang[b], Lixin Gao[a]

[a] University of Massachusetts Amherst, 151 Holdsworth Way, Amherst, MA 01003, USA
[b] Northeastern University, 11 Wenhua Road, Shenyang, Liaoning 110819, China

HIGHLIGHTS

- Propose two approaches to parallelize EM algorithms with frequent updates in a distributed environment to scale to massive data sets.
- Prove that both approaches maintain the convergence properties of the EM algorithms.
- Design and implement a distributed framework to support the implementation of frequent updates for the EM algorithms.
- Extensively evaluate our framework on both a cluster of local machines and the Amazon EC2 cloud with several popular EM algorithms.
- The EM algorithms with frequent updates implemented on our framework can converge much faster than traditional implementations.

ARTICLE INFO

Article history:
Received 29 December 2016
Received in revised form 1 June 2017
Accepted 20 July 2017
Available online 29 July 2017

Keywords:
Expectation–Maximization
Frequent updates
Concurrent updates
Distributed framework
Clustering
Topic modeling

ABSTRACT

Expectation–Maximization (EM) is a popular approach for parameter estimation in many applications, such as image understanding, document classification, and genome data analysis. Despite the popularity of EM algorithms, it is challenging to efficiently implement these algorithms in a distributed environment for handling massive data sets. 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 two categories of EM applications, clustering and topic modeling. These applications include k-means clustering, fuzzy c-means clustering, parameter estimation for the Gaussian Mixture Model, and variational inference for Latent Dirichlet Allocation. We extensively evaluate our framework on both a cluster of local machines and the Amazon EC2 cloud. Our evaluation shows that the EM algorithms with frequent updates implemented on FreEM can converge much faster than those implementations with traditional concurrent updates.

1. 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, and genome data analysis. Expectation–Maximization (EM) [8] is one of the most popular approaches in these applications [5,9,32]. 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 the popularity, many methods for accelerating EM algorithms have been proposed. Some of them [18,21] 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 to allow 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, which can be computed

[☆] Part of this work has been published in Proceedings of CLUSTER'12: 2012 IEEE International Conference on Cluster Computing (Yin et al., 2012).

[*] Corresponding author.

E-mail addresses: jyin@ecs.umass.edu (J. Yin), zhangyf@cc.neu.edu.cn (Y. Zhang), lgao@ecs.umass.edu (L. Gao).

http://dx.doi.org/10.1016/j.jpdc.2017.07.005
0743-7315/© 2017 Elsevier Inc. All rights reserved.

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 considerable 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 updates to achieve 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 updates. 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 updates. We prove that both approaches maintain the convergence properties of 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 evaluate FreEM in the context of a wide class of well-known EM applications: k-means clustering, fuzzy c-means clustering, parameter estimation for the Gaussian Mixture Model, and Latent Dirichlet Allocation for topic modeling. Our results show that the EM algorithm with frequent updates can run much faster than that with traditional concurrent updates. In addition, EM algorithms can be implemented on FreEM in a more efficient way than on Hadoop [13], an open source implementation of the popular programming model MapReduce [7].

The rest of this paper is organized as follows. Section 2 describes the EM algorithm with frequent updates. Section 3 exemplifies frequent updates through EM applications. Section 4 presents our approaches to parallelize the EM algorithm with frequent updates. In Section 5, we present the design, implementation and API of FreEM. Section 6 is devoted to the evaluation results. Finally, we discuss related work in Section 7 and conclude the paper in Section 8.

2. EM algorithms

In a statistical model, suppose that we have observed the value of one random variable, X, which comes from a parameterized family, $P(X|\theta)$. The value of another variable, Z, is hidden. Given 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 $\{x_1, \ldots, x_n\}$, and thereby the hidden variable can be decomposed as $\{Z_1, Z_2, \ldots, Z_n\}$. 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, an EM algorithm is usually 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 leverages the data points and the current estimates of the parameters to estimate the distribution of hidden variables. The M-step updates the parameters to be those maximizing the likelihood found in the E-step.

One classic example of the EM algorithm is k-means clustering [16]. It aims to partition n data points $\{x_1, x_2, \ldots, x_n\}$ into k $(k \leq n)$ clusters $\{c_1, c_2, \ldots, c_k\}$ so as to minimize the objective function:

$$f = \sum_{i=1}^{k} \sum_{x_j \in c_i} \|x_j - \mu_{c_i}\|^2, \qquad (2.1)$$

where $\mu_{c_i} = \frac{1}{|c_i|} \sum_{x_j \in c_i} x_j$ is the centroid of cluster c_i.

The E-step of k-means assigns points to the cluster with the closest mean. That is, a data point x_j is assigned to cluster c if $c = \arg\min_j \|x_i - \mu_{c_j}\|^2$. Its M-step updates the centroids (parameters) for all clusters.

2.1. 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.

The vanilla k-means (Lloyd's algorithm [15]) belongs to this category. Its E-step performs cluster assignment for each data point, and it M-step updates the centroids along the direction of minimizing the objective function.

2.2. The EM algorithm with frequent updates

The EM algorithm with frequent updates attempts to accelerate the convergence by frequently updating the parameters. This algorithm can provide more up-to-date parameters to process data points and to potentially speedup convergence. However, updating parameters frequently may incur significant overhead if the update is done in the original way. In order to conquer this obstruction, we introduce a way of updating parameters incrementally. In the EM algorithm, the distribution influences the likelihood of the parameters via some sufficient statistics. The statistics is usually the summation over the statistics on each individual data point, and a summation can be incrementally updated. As a result, the cost of computing the sufficient statistics grows linearly with the number of data points whose statistics have been changed in the E-step.

Take k-means for instance. 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:

$$\begin{array}{ll} S_c = S_c - x_i, & S_{c'} = S_{c'} + x_i; \\ W_c = W_c - 1, & W_{c'} = W_{c'} + 1. \end{array} \tag{2.2}$$

We can see that the cost of computing S_i and W_i grows linearly with the number of data points whose cluster assignments have been changed.

Nevertheless, performing frequent updates 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 partition data points into mutually disjoint blocks and iterates through the blocks in a cyclic way. Each iteration processes a block of data points in the E-step and then perform the M-step immediately to update the parameters. 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. One iteration of the algorithm 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 $\sum_{i=1}^{n} E_{Q_i}[\log P(x_i, z_i|\theta)]$.

Take k-means as an example. With update by block, it selects a block of data points to perform the E-step, and followed the M-step. The E-step for each data point is the same as before, and M-step still involves all data points. Then, it selects another block to repeat the steps.

The other one is *update by subrange*, which computes 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 $\sum_{i=1}^{n} E_{Q_i}[\log P(x_i, z_i|\theta)]$.

Again take k-means as an example. With update by subrange, its E-step performs cluster assignments among a subset of clusters (e.g., s clusters, $s \le k$), and its M-step updates of the centroids of the clusters in this subset. Then, it selects another subset to repeat the steps.

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. Furthermore, even the combined version maintains the convergence properties of the EM algorithm, as stated in the following theorem.

Theorem 2.1. *An EM algorithm with frequent updates converges.*

The rationale behind Theorem 2.1 is that like an original EM algorithm, an EM algorithm with frequent updates monotonically increases the log likelihood function $L(\theta)$ in the sequence of E-steps and M-steps. The formal proof is provided in Appendix.

3. Applications of the EM algorithm

In this section, we describe two categories of EM applications, clustering and topic modeling. In the clustering category, we illustrate k-means clustering, Fuzzy c-means clustering, parameter estimation for the Gaussian Mixture model. In the topic modeling category, we discuss variational inference for Latent Dirichlet Allocation. We illustrate how to incrementally compute the statistics and how to derive the parameters from the statistics. 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. Furthermore, we analyze the time complexity and the space complexity of frequent updates to show the feasibility and efficiency of frequent updates from the algorithmic perspective.

3.1. Clustering

Clustering is one of the most important tasks of data mining. It has been leveraged in many fields, including pattern recognition, image analysis, information retrieval, and bioinformatics.

3.1.1. K-means

We have illustrated k-means in the above section as a running example for the EM algorithm. Here we analyze the space complexity and the time complexity of k-means with frequent updates. In order to perform incremental computation, we need to store cluster assignments for all data points and the statistics S_c and W_c, which only take $O(n + kd)$ space, where d is the dimension of a data point (while storing data points in memory takes $O(nd)$ space). We next analyze the complexity of frequent updates. Take the update by block method for example. Suppose data points are equally split into b blocks (with each block having n/b data points). Performing the E-step on one block takes $O(nkd/b)$ time, since processing one data point takes $O(kd)$ time. The following M-step takes $O(nd/b + kd)$ time, in which updating statistics S_c and W_c needs $O(nd/b)$ time and deriving all centroids from the statistics (*e.g.*, $\mu_i = \frac{S_i}{W_i}$) takes $O(kd)$ time. As a result, processing all data points in one pass (including multiple E-steps and M-steps) requires $O(nkd + bkd)$ time. Since $b \le n$, the time can be represented as $O(nkd)$. Furthermore, we can show the original k-means (i.e., k-means with concurrent updates) also needs $O(nkd)$ time to process all data points in one pass. In other words, with incremental computation, the update by block approach will not increase the time complexity no matter how frequent the M-step is performed. Similarly, the update by subrange approach also does not increase the time complexity of E-steps, and the time complexity of M-steps can be ignorable compared to that of E-steps. Therefore, the same conclusion can be obtained for the update by subrange approach.

3.1.2. Fuzzy C-means

Given a set of data points $\{x_1, x_2, \ldots, x_n\}$, fuzzy c-means (FCM) [3,10] aims to assign n data points into C clusters $\{c_1, c_2, \ldots, c_k\}$ so as to minimize the objective function:

$$J_m = \sum_{i=1}^{n} \sum_{j=1}^{C} \mu_{ij}^m \|x_i - c_j\|^2, \tag{3.1}$$

where m ($m > 1$) is the fuzzy factor, μ_{ij} is the degree of membership of x_i belonging to cluster j, and c_j is the centroid of cluster j.

The degree of membership μ_{ij} and the centroid c_j are computed by the equations:

$$\mu_{ij} = \frac{1}{\sum_{k=1}^{C}(\frac{\|x_i-c_j\|}{\|x_i-c_k\|})^{\frac{2}{m-1}}}, \qquad c_j = \frac{\sum_{i=1}^{N}\mu_{ij}^{m}x_i}{\sum_{i=1}^{N}\mu_{ij}^{m}}. \tag{3.2}$$

If we describe FCM in the EM setting, its E-step updates the degree of membership for all data 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 membership to cluster j changes from μ_{ij} to μ'_{ij}, the statistics can be incrementally updated as follows:

$$W_j = W_j - (\mu_{ij})^m + (\mu'_{ij})^m, \qquad X_j = X_j + ((\mu'_{ij})^m - (\mu_{ij})^m)x_i. \tag{3.3}$$

We now analyze the space complexity and the time complexity of FCM with frequent updates. In order to perform incremental computation, we need to store the degree of membership for all data points and the statistics W_j and X_j, which take $O(kn + kd)$ space. Similar to the time complexity analysis for k-means, we can show that processing all data points in one pass (including multiple E-steps and M-steps) requires $O(nkd)$ time for the update by block approach. Furthermore, original FCM also needs $O(nkd)$ time to process all data points in one pass. As a result, FCM with frequent updates does not increase the time complexity.

The details of fitting Gaussian Mixture Model (GMM) in the EM setting and incrementally updating the statistics for GMM can be found in [28], and thus skipped here due to space limitations.

3.2. Topic modeling

An EM algorithm is also a powerful tool for statistical text analysis, such as topic modeling. Topic modeling provides a way to navigate large document collections by discovering the themes that permeate a corpus. In particular, Latent Dirichlet Allocation (LDA) [4] is a popular topic modeling approach. It provides a generative model that describes how the documents in a corpus were produced. First, we denote the M given documents represented as $d_1, d_2, \dots, d_M$. Let V denote the number of words in the vocabulary, and let N_i represent the number of words in a document d_i. Moreover, we use w_j to denote the jth word in the vocabulary and $w_{i,j}$ to represent the jth word in the i th document. Assume that the documents are represented as random mixtures over K topics. A topic is a K dimensional multinomial distribution over words, and the i th topic is denoted as ϕ_i. We use θ_i to represent the topic distribution for a document d_i. Furthermore, assume $w_{i,j}$ is drawn form topic $z_{i,j}$. In addition, we use α and β to represent hyper parameters of the Dirichlet distribution. LDA assumes the following generative process.

1. For each topic index $k \in \{1, \dots, K\}$, draw topic distribution $\phi_k \sim Dir(\beta)$.
2. For each document $d_i \in \{d_1, d_2, \dots, d_M\}$:
    - Draw topic distribution $\theta_i \sim Dir(\alpha)$.
    - For $j \in \{1, 2, \dots, N_i\}$
        - Draw $z_{i,j} \sim Mult(\theta_i)$.
        - Draw $w_{i,j} \sim Mult(\phi_{z_{i,j}})$.

In the process, $Dir()$ denotes a Dirichlet distribution, and $Mult()$ represents a multinomial distribution.

There are two widely used approximate inference techniques for LDA. One is Markov chain Monte Carlo (MCMC) sampling (e.g., Gibbs sampling) [11], and the other one is variational inference [4]. Even though MCMC is a powerful methodology, the convergence of the sampler to its stationary distribution is usually hard to diagnose, and sampling algorithms may converge slowly in high dimensional models. Variational inference methods have clear convergence criterion and provide efficiency advantages over sampling techniques in high dimensional problems [20].

The basic idea of variational inference is to leverage Jensen's inequality to obtain an adjustable lower bound on the log likelihood of the posterior distribution. We refer interesting readers to original LDA paper [4] for more details. The variational inference aims to maximize the following likelihood (i.e., the objective function) [4]:

$$\begin{aligned} L(\gamma, \phi; \alpha, \beta) &= \log \Gamma\left(\sum_{j=1}^{K}\alpha_j\right) - \sum_{i=1}^{K}\log\Gamma(\alpha_i) \\ &+ \sum_{i=1}^{K}(\alpha_i - 1)\left(\Psi(\gamma_i) - \Psi\left(\sum_{j=1}^{K}\gamma_j\right)\right) \\ &+ \sum_{n=1}^{N}\sum_{i=1}^{K}\phi_{ni}\left(\Psi(\gamma_i) - \Psi\left(\sum_{j=1}^{K}\gamma_j\right)\right) \\ &+ \sum_{n=1}^{N}\sum_{i=1}^{K}\sum_{j=1}^{V}\phi_{ni}w_{nj}\log\beta_{ij} - \log\Gamma\left(\sum_{j=1}^{K}\gamma_j\right) \\ &+ \sum_{i=1}^{K}\log\Gamma(\gamma_i) - \sum_{i=1}^{K}(\gamma_i - 1)\left(\Psi(\gamma_i) - \Psi\left(\sum_{j=1}^{K}\gamma_j\right)\right) \\ &- \sum_{n=1}^{N}\sum_{i=1}^{K}\phi_{ni}\log\phi_{ni}, \end{aligned} \tag{3.4}$$

where the Dirichlet parameter γ and the multinomial parameters ϕ_n are the free variational parameters, $\Gamma()$ is the Gamma function and $\Psi()$ is the first derivative of $\log\Gamma()$.

One popular method to minimize the Kullback–Leibler divergence (i.e., to maximize the above objective function) is to use an EM approach. Variational EM alternates between updating the expectations of the variational distribution q and maximizing the probability of the parameters given the observed documents. Here each document is one data point.

The M-step of variational EM updates α using a Newton–Raphson method. For ease of exposition, we assume all elements of α are the same unless otherwise stated, and thus α can be simply a single value in the following updates. Updates are carried out in log-space [4], as follows:

$$\log(\alpha^{t+1}) = \log(\alpha^{t}) - \frac{\partial L}{\partial\alpha} \Big/ \left(\frac{\partial^2 L}{\partial\alpha^2}\alpha + \frac{\partial L}{\partial\alpha}\right), \tag{3.5}$$

$$\begin{aligned} \frac{\partial L}{\partial\alpha} &= M(K\Psi(K\alpha) - K\Psi(\alpha)) \\ &\quad + \sum_{d=1}^{M}\left(\sum_{i=1}^{K}\Psi(\gamma_{di}) - K\Psi\left(\sum_{j=1}^{K}\gamma_{dj}\right)\right), \end{aligned} \tag{3.6}$$

$$\frac{\partial^2 L}{\partial\alpha^2} = M(K^2\Psi'(K\alpha) - K\Psi'(\alpha)). \tag{3.7}$$

From Eqs. (3.5)–(3.7) we can see that only the second part of $\frac{\partial L}{\partial\alpha}$ depends on each individual document. Therefore, in order to incrementally update α, we let

$$R = \sum_{d=1}^{M}\left(\sum_{i=1}^{K}\Psi(\gamma_{di}) - K\Psi\left(\sum_{j=1}^{K}\gamma_{dj}\right)\right),$$
$$s_d = \sum_{i=1}^{K}\Psi(\gamma_{di}) - K\Psi\left(\sum_{j=1}^{K}\gamma_{dj}\right). \quad (3.8)$$

When updating a document, i, if its s_d changes from s_i' to s_i, we can incremental update the statistics R in the way, $R = R + s_i - s_i'$.

Next, we illustrate how to update β incrementally. We have (with skipping the step of normalizing β_i for simplicity) $\beta_{ij} = \sum_{d=1}^{M}\sum_{n=1}^{N_d}\phi_{dni}w_{dnj}$.

One simple way to perform incremental updates is to cache ϕ_{dni}. Then when a document changes ϕ_{dni}' to ϕ_{dni}, we can update β_{ij} in the way, $\beta_{ij} = \beta_{ij} + \sum_{n=1}^{N_d}(\phi_{dni} - \phi_{dni}')w_{dnj}$. However, caching ϕ_{dni} for all documents takes $O(MKV)$ space, which can be huge. In order to address the space issue, we present a space-efficient incremental scheme, which is suitable for the update by block approach. We divide documents into b blocks, $\{B_1, B_2, \ldots, B_b\}$. Let

$$\beta_{ij}^{(l)} = \sum_{d\in B_l}\sum_{n=1}^{N_d}\phi_{dni}w_{dnj}. \quad (3.9)$$

Then, we have

$$\beta_{ij} = \sum_{l=1}^{b}\beta_{ij}^{(l)}. \quad (3.10)$$

When the documents in block l are updated, we compute $\beta_{ij}^{(l)}$ from scratch with Eq. (3.9), and then recover β_{ij} using Eq. (3.10). In this way, we only need to cache $\beta_{ij}^{(l)}$, $1 \le l \le b$. When b is small (e.g., a constant less than 10), then caching only takes $O(KV)$ space.

Furthermore, we can show that LDA with frequent updates (e.g., the update by block approach) needs $O(KV)$ space to cache R, $\beta_{ij}^{(l)}$, and β_{ij} in order to support incremental computation. Additionally, performing the E-step on one document takes $O(IKV)$ time, where I the number of iterations the E-step needs to converge on the document. With incremental computation, if there are m documents updated in the E-step, the following M-step takes $O(mKV)$ time. As a result, LDA with frequent updates requires $O(MIKV)$ time to process all documents in one pass (including multiple E-steps and M-steps). Furthermore, original LDA also needs $O(MIKV)$ time to process all documents in one pass. Consequently, LDA with frequent updates does not increase the time complexity.

4. Parallelizing frequent updates

Parallelizing frequent updates in a distributed environment is challenging. Although computing the distribution and incrementally updating the local statistics 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 data 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 briefly 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 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.

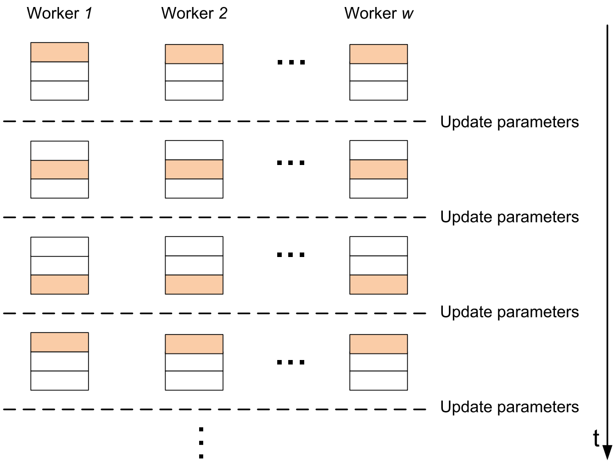


Fig. 1. Process of the partial concurrent method. The colored box indicates the picked block of data points for computing the distribution.

4.1. Concurrent method

In the traditional method of parallelizing concurrent updates, 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 this method as *concurrent* method.

4.2. 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 2. 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 Fig. 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 new 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 in a round-robin fashion to guarantee each data point has an equal chance to be updated. In other words, first m data points are selected as one block, and then second m data points, and so on. 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). Furthermore, either when each worker works individually to compute the distribution or when all workers synchronize to derive the new parameters, the objective function keeps increasing (or decreasing, we assume "increasing" for brevity in this section). Therefore, we have the following theorem.

Theorem 4.1. *The partial concurrent method maintains the convergence property of an EM algorithm.*

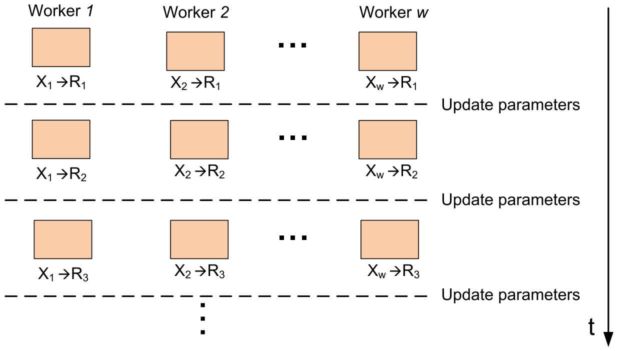


Fig. 2. Process of the subrange concurrent method. Each worker recomputes the distribution among the subrange (R_i) for all of its local data points (X_j).

The size of the block (*i.e.*, m) plays an important role on the efficiency of the partial concurrent method. It indicates the tradeoff 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 discussion of the optimal block size can be found in [28]. Our framework provides a recommended block size.

4.3. Subrange concurrent method

Our second method to parallelize the EM algorithm with frequent updates corresponds to the update by subrange approach in Section 2. Recall that the update by subrange approach recomputes the distribution over the subrange of hidden variables. As shown in Fig. 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). Furthermore, either when each worker computes the distribution among the subrange or when all workers synchronize to derive the new parameters, the objective function keeps increasing. Therefore, we have the following theorem.

Theorem 4.2. *The subrange concurrent method maintains the convergence property of an EM algorithm.*

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 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 the 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. The discussion of the optimal subrange size can be found in [28] as well.

5. FreeEM

MapReduce [7] and its variants [12,27] have emerged as popular distributed frameworks for data intensive computation. However, MapReduce does not support EM algorithms well, due to its inefficiency in supporting iterative processes. 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 [30]. 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.

5.1. 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 data 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*.

5.2. 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. More details of the implementation of the framework and its APIs can be found in [28].

5.3. 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 effect the performance of the algorithm. For the partial concurrent method, we use an empirical approach to find the partial concurrent method, which minimizes the workload of reaching a convergence point. We use the same idea to figure out the optimal subrange size. More details of determining the optimal block size and the optimal subrange size can be found in [28].

Table 1
Data sets for clustering.

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

Table 2
Data sets for topic modeling.

Data Set	# Documents	# Unique words	# Total words
KOS	3,430	6,906	467,714
Enron	39,861	28,102	6,400,000
NYTimes	300,000	102, 660	100,000,000

6. Evaluation

In this section, we evaluate the effectiveness and efficiency of EM algorithms with frequent updates on a single machine and in a distributed environment. All the applications described in Section 3 are tested. For the distributed environment, all the parallel methods, including concurrent, partial concurrent, and subrange concurrent, are evaluated on FreEM. We also compare the concurrent method on FreEM with that on Hadoop.

6.1. Experiment setup

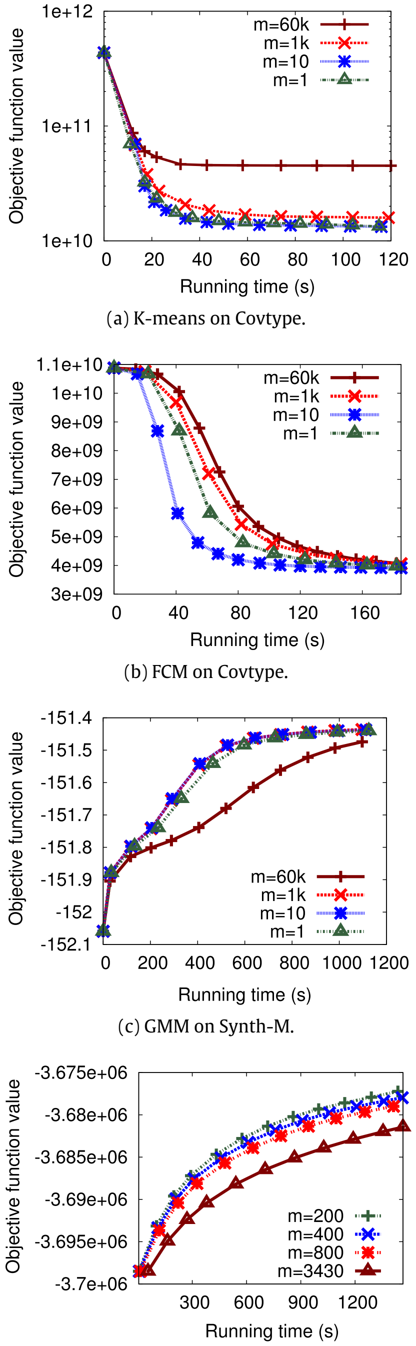
We build a small-scale cluster of local machines and a large-scale cluster on Amazon EC2 [1]. The small-scale cluster consists of 4 machines, and each one has a dual-core 2.66 GHz CPU, 4 GB of RAM, 1TB of disk. These 4 machines are connected through a switch with a bandwidth of 1 Gbps. The Amazon cluster consists of 40 medium instances, each of which having 2 EC2 compute units, 3.75 GB of RAM, and 400 GB of hard disk.

Real-world data sets from UCI Machine Learning Repository [22] and synthetic data sets are leveraged. The synthetic data sets are generated in such a way: each dimension (i.e., attribute) of one data point follows a Gaussian distribution with random mean and standard deviation 1.0. The data sets are summarized in Tables 1 and 2.

6.2. Single machine experiments

We first demonstrate the advantages of the EM algorithm with frequent updates on one single machine. The update by block approach is used as an example. All the EM applications described in Section 3 are implemented. It is worthy to note that k-means and FCM aim to minimize their objective functions, while GMM and LDA aim to maximize their objective functions. All the four objective functions have been shown in either Section 2 (k-means) or Section 3 (FCM, GMM, LDA).

First, we perform the three clustering applications, k-means, FCM, and GMM, with various block size (m). For a fair comparison, each application runs on one data set with the same initial start. Data sets sampled from the original data sets are used to evaluate the clustering applications. Each data set consists of 60,000 data points. We sample the data sets since a single commodity machine cannot hold the whole data set in memory. Figs. 3(a)–3(c) presents the convergence speed. As shown, the EM algorithm with frequent updates ($m < 60k$) converges faster and may achieve a better convergence point, compared to that with concurrent updates ($m = 60k$). These figures also demonstrate the update frequency (determined by the block size) has a significant impact on the performance. Then, we perform LDA on the KOS data set. Fig. 3(d) plots the convergence speed with different block sizes.



(a) K-means on Covtype.

(b) FCM on Covtype.

(c) GMM on Synth-M.

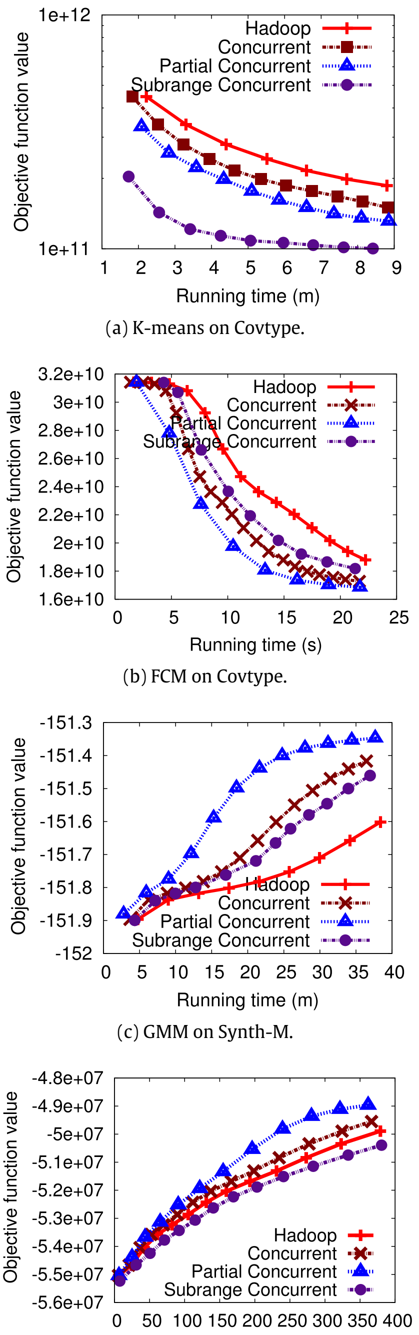
(d) LDA on KOS.

Fig. 3. Convergence speed on the single machine. For K-means, FCM, and GMM, $m = 60k$ corresponds to concurrent updates; for LDA, $m = 3430$ corresponds to concurrent updates.

They further show that the EM algorithm with frequent updates converge faster than that with concurrent updates ($m = 3430$) and that the update frequency effects the performance.

6.3. Small-scale cluster experiments

FreEM allows the EM algorithm to frequently update the parameters in a distributed environment 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



(a) K-means on Covtype.

(b) FCM on Covtype.

(c) GMM on Synth-M.

(d) LDA on Enron.

Fig. 4. Convergence speed on the small-scale cluster.

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 Hadoop is a popular framework, we utilize the convergence speed of the concurrent method on it as the base line.

The convergence speed evaluation is first performed on the local cluster. All the methods start with the same initial setting, when compared on the same data set. We set the number of clusters as 80 for all experiments of clustering applications. Figs. 4(a)–4(c) show the performance comparison. Note that the vertical axes not starting at 0 is because we zoom out to focus on the performance close to the convergence point. First, we examine the improvement from the algorithmic perspective. We can see that the partial concurrent method converges 1.3x–1.7x faster than the concurrent method for all the three clustering applications. The subrange concurrent method converges 3.4x faster and converges to a much better point than the concurrent method for k-means. Unfortunately, the subrange concurrent method is slower than the concurrent method on FCM and GMM. Then, we examine the improvement from the framework perspective. The convergence speed of the concurrent method on FreEM is 1.4x–1.6x 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 [30,31]. 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 applications. For LDA, we set the number of topics as 100. From Fig. 4(d), we can see that the partial concurrent method converges 1.4x faster than the concurrent method.

6.4. *Large-scale cluster experiments*

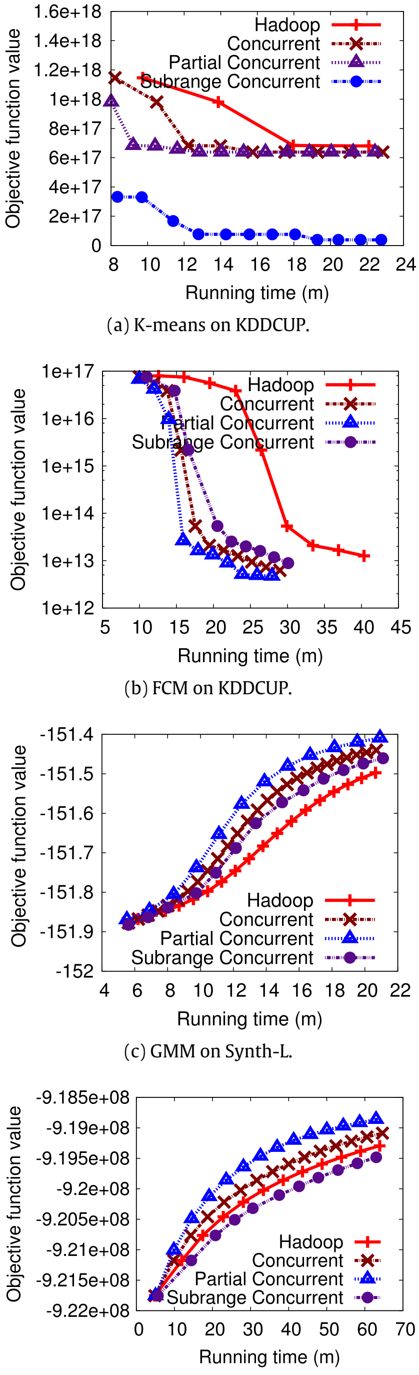
In order to validate the scalability of FreEM, we also evaluate it on the Amazon EC2 cloud. We first show the performance comparison when all the 40 instances are used. From Fig. 5, we can see that the partial concurrent method converges faster than the concurrent method for all the EM applications and that the subrange concurrent method converges 1.2x–1.4x faster and converges to a much better point than the concurrent method for k-means.

We then evaluate the scaling performance of FreEM as the number of workers increases from 10 to 40. The speedup is measured relative to the running time of 10 workers. Here the running time means the wall clock time that an EM application takes to reach a pre-defined threshold. 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.

Figs. 6 and 7 show that both the concurrent method and the partial concurrent method exhibit good speedups. The former one 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 Fig. 6(b), 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, as plotted in Fig. 7(a). However, the subrange concurrent method still runs much faster than the concurrent method even on 40 workers, as shown in Fig. 7(b).

7. 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. [8] and Meng et al. [17] present a partial M-step may accelerate the algorithm



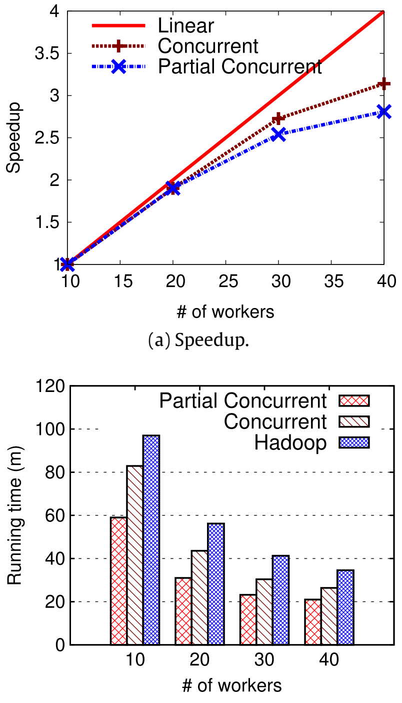
(a) K-means on KDDCUP.

(b) FCM on KDDCUP.

(c) GMM on Synth-L.

(d) LDA on NYTimes.

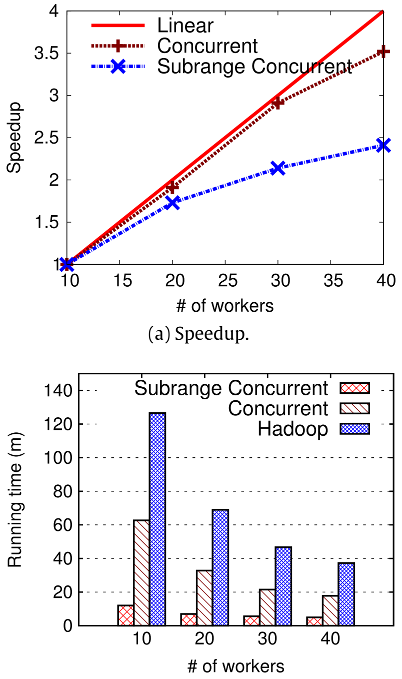
Fig. 5. Convergence speed on the Amazon EC2 cloud.



(a) Speedup.

(b) Performance comparison.

Fig. 6. Scaling performance of the partial concurrent method.



(a) Speedup.

(b) Performance comparison.

Fig. 7. Scaling performance of the subrange concurrent method.

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. [18] and Thiesson et al. [21] 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. [18] prove that such a variant of the EM algorithm converges. Thiesson et al. [21] 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. Furthermore, these two pieces of work demonstrate the power of frequent update through only one EM application, parameter estimation for a finite mixture model, whereas our work covers much broader applications.

There are a number of efforts targeted on parallelizing the EM algorithm as well. Araújo et al. [2] present a parallelized implementation of GMM on GPUs. Plant et al. [19] introduces a parallel variant of the EM algorithm which allows asynchronous model updates. Cui et al. [6] investigate implementations of GMM on different distributed frameworks. There are also a group of researches focusing on efficiently updating the parameters in the M-step. For examples, Wolfe et al. [23] propose an approach to distribute both the E-step and the M-step based on MapReduce. Kowalczyk et al. [14] present a gossip-based distributed implementation of the EM algorithm for GMM. Zhai et al. [29] introduce a MapReduce-based implementation of the EM algorithm for LDA. 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.

Frequent updates also show promising results in other algorithms, such as nonnegative matrix factorization [24]. More generally, frequent updates can be considered as one form of iterative computation transformation, which aims to accelerate large-scale data processing. Iterative computation transformation can be applied to a wide range of algorithms, such as belief propagation [25] and graph algorithms [26].

Part of this work has been published in a conference [28]. Compared to the previous version, this paper extends the EM algorithm with frequent updates to one essential and complex application in data mining, LDA for topic modeling. In this paper, we introduce the new challenge of applying frequent updates to LDA and present our solution. Furthermore, we implement LDA on FreEM and evaluate it with real-word data sets on both a local cluster and the Amazon EC2 cloud. We also analyze the time space complexities of frequent updates for EM applications in this paper. By doing this, we can show the feasibility and efficiency of frequent updates from the algorithm-wise view. Moreover, we add the complete proof which shows an EM algorithm with frequent updates converges and add the proof that shows our methods of parallelizing frequent updates in a distributed environment maintain the convergence properties.

8. Conclusion

Motivated by the observations that the EM algorithm performing frequent updates is much more efficient than it performing 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. Furthermore, we formally 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 two categories of EM applications, clustering and topic modeling. The evaluation results show that the EM algorithm with frequent updates can run much faster than the one with traditional concurrent updates. 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

This work is partially supported by National Science Foundation grants CNS-1217284 and CCF-1018114, Natural Science Foundation of China (61528203, 61672141). Any opinions, findings, conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the sponsor.

Appendix

The following is the proof for Theorem 2.1. We first consider the following derivation:

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

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

$$J(Q, \theta) = \sum_{i=1}^{n} \sum_{z_i} Q_i(z_i) \log \frac{P(x_i, z_i|\theta)}{Q_i(z_i)}, \tag{A.2}$$

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 increase $J(Q, \theta)$, it will converge to a local maximum (or a saddle point) of $L(\theta)$. For simplicity, we ignore the possibility that it converges to a saddle point. Next, we will prove that each iteration of the EM algorithm with frequent updates either improves $J(Q, \theta)$ or leaves it unchanged, and thus it converges a local maximum of $L(\theta)$. For this purpose, we are going to introduce Lemmas A.1–A.3, and Theorem A.4.

Lemma A.1. *Given a fixed value of θ, for each i, there is a unique distribution, $Q_i(\cdot)$, that maximizes $J(Q, \theta)$, achieved by $Q_i(z_i) = P(z_i|x_i, \theta)$. Moreover, the $Q_i(z_i)$ varies continuously with θ.*

Proof of Lemma A.1. We need to prove that for any i, $Q_i(z_i) = P(z_i|x_i, \theta)$ maximizes $\sum_{z_i} Q_i(z_i) \log \frac{P(x_i, z_i|\theta)}{Q_i(z_i)}$ with respect to $Q_i(\cdot)$. We know $\sum_{z_i} Q_i(z_i) = 1$. Therefore, the maximum can be found using a Lagrange multiplier. At such a maximum, we will have $Q_i(z_i) \propto P(x_i, z_i|\theta)$. Note that $\sum_{z_i} Q_i(z_i) = 1$. We have the unique solution $Q_i(z_i) = \frac{P(x_i, z_i|\theta)}{\sum_{z_i} P(x_i, z_i|\theta)} = P(z_i|x_i, \theta)$. Consequently, given a fixed value of θ, for each i, if $Q_i(z_i) = P(z_i|x_i, \theta)$, $J(Q, \theta)$ is maximized. Since $P(z_i|x_i, \theta)$ varies continuously with θ, $Q_i(z_i)$ varies continuously with θ.

Lemma A.2. *If $Q_i(z_i) = P(z_i|x_i, \theta)$ for each i, $L(\theta) = J(Q, \theta)$.*

Using Jensen's inequality, it is straightforward to prove Lemma A.2, so we skip the formal proof here.

Lemma A.3. *If $J(Q, \theta)$ has a local maximum at Q^* and θ^*, then a local maximum of $L(\theta)$ occurs at θ^* as well.*

Proof of Lemma A.3. From Lemmas A.1 and A.2, we see that if $Q_i(z_i) = P(z_i|x_i, \theta)$ for each i, then $L(\theta) = J(Q, \theta)$ for any θ. Therefore, $L(\theta^*) = J(Q^*, \theta^*)$, where Q^* means $Q_i(z_i) = P(z_i|x_i, \theta^*)$ for each i. To show that a local maximum of $L(\theta)$ occurs at θ^*, we need to show that there is no θ' near to θ^* which lets $L(\theta') > L(\theta^*)$. If such a θ' existed, we would have $J(Q', \theta') > J(Q^*, \theta^*)$, where Q' means $Q_i(z_i) = P(z_i|x_i, \theta')$ for each i. From Lemma A.1, we know that Q varies continuously with θ. Therefore, Q' must be near to Q^*. However, it contradicts that $J(Q, \theta)$ has a local maximum at Q^* and θ^*.

Theorem A.4. *The EM algorithm with frequent updates converges to a local maximum of $L(\theta)$.*

Proof of Theorem A.4. Let $F_i(x_i, Q_i, \theta) = \sum_{z_i} Q_i(z_i) \log \frac{P(x_i, z_i|\theta)}{Q_i(z_i)}$, then $J(Q, \theta) = \sum_{i=1}^{n} F_i(x_i, Q_i, \theta)$. In the E-step of the EM algorithm with frequent updates, we change the value of $F_i(x_i, Q_i, \theta)$ for a subset of data points (e.g., S_m) through changing $Q_i(\cdot)$. If we can show $F_e(x_e, Q_e^{(t)}, \theta) \geq F_e(x_e, Q_e^{(t-1)}, \theta)$ for any $x_e \in S_m$, then we prove the E-step increase $J(Q, \theta)$. Assume that we wish to maximize $\sum_{z_e \in B} Q_e(z_e) \log \frac{P(x_e, z_e|\theta)}{Q_e(z_e)}$ respect to Q_e. We also know $\sum_{z_e \in B} Q_e(z_e) = c_B$ (c_B is constant number). The maximum can be found using a Lagrange multiplier (maximize $\sum_{z_e \in B} Q_e(z_e) \log \frac{P(x_e, z_e|\theta)}{Q_e(z_e)}$, subject to $\sum_{z_e \in B} Q_e(z_e) = c_B$). At such a maximum, we will have $Q_e(z_e) \propto P(x_e, z_e|\theta)$ (for $z_e \in c_B$). Note that $\sum_{z_e \in c_B} Q_e(z_e) = c_B$. We have the unique solution $Q_e(z_e) = \frac{P(x_e, z_e|\theta) * c_B}{\sum_{z_e} P(x_e, z_e|\theta)} = P(z_e|x_e, \theta) * c_B$ (for $z_e \in c_B$). Therefore, the E-step increases $F_e(x_e, Q_e, \theta^{(t-1)})$ by setting $Q_e(z_e) = P(z_e|x_e, \theta^{(t-1)})$. Consequently, it increases $J(Q, \theta)$. The M-step of the EM algorithm with frequent updates obtain $\theta^{(t)}$ by maximizing $J(Q, \theta)$. Therefore, the M-step increases $J(Q, \theta)$ as well. Since both its E-step and its M-step increase $J(Q, \theta)$, the EM algorithm with frequent updates converges to a local maximum of $J(Q, \theta)$. By combining with Lemma A.3, we know that the EM algorithm with frequent updates converges to a local maximum of $L(\theta)$.

References

[1] Amazon elastic compute cloud (Amazon EC2), http://aws.amazon.com/ec2/.

[2] G.F. Araújo, H.T. Macedo, M.T. Chella, C.A.E. Montesco, M.V.O. Medeiros, Parallel implementation of expectation-maximisation algorithm for the training of Gaussian mixture models, J. Comput. Sci. 10 (10) (2014) 2124–2134.

[3] J.C. Bezdek, Pattern Recognition with Fuzzy Objective Function Algorithms, Kluwer Academic Publishers, 1981.

[4] D.M. Blei, A.Y. Ng, M.I. Jordan, Latent dirichlet allocation, J. Mach. Learn. Res. 3 (2003) 993–1022.

[5] X. Cheng, S. Su, L. Gao, J. Yin, Co-ClusterD: A distributed framework for data co-clustering with sequential updates, IEEE Trans. Knowl. Data Eng. 27 (12) (2015) 3231–3244.

[6] H. Cui, J. Wei, W. Dai, Parallel implementation of expectation-maximization for fast convergence, https://users.ece.cmu.edu/~hengganc/archive/report/final.pdf.

[7] J. Dean, S. Ghemawat, MapReduce: Simplified data processing on large clusters, in: OSDI '04, 2004.

[8] A.P. Dempster, N.M. Laird, D.B. Rubin, Maximum likelihood from incomplete data via the EM algorithm, J. R. Stat. Soc. Ser. B Stat. Methodol. 39 (1) (1977).

[9] G. Di Fatta, F. Blasa, S. Cafiero, G. Fortino, Fault tolerant decentralised k-means clustering for asynchronous large-scale networks, J. Parallel Distrib. Comput. 73 (3) (2013) 317–329.

[10] J.C. Dunn, A fuzzy relative of the ISODATA process and its use in detecting compact well-separated clusters, J. Cybern. 3 (3) (1973) 32–57.

[11] T.L. Griffiths, M. Steyvers, Finding scientific topics, Proc. Natl. Acad. Sci. USA 101 (suppl. 1) (2004) 5228–5235.

[12] R. Gu, X. Yang, J. Yan, Y. Sun, B. Wang, C. Yuan, Y. Huang, SHadoop: Improving MapReduce performance by optimizing job execution mechanism in Hadoop clusters, J. Parallel Distrib. Comput. 74 (3) (2014) 2166–2179.

[13] Hadoop, http://hadoop.apache.org/.

[14] W. Kowalczyk, N. Vlassis, Newscast EM, in: NIPS '04, pp. 713–720.

[15] S. Lloyd, Least squares quantization in PCM, IEEE Trans. Inf. Theory 28 (2) (1982) 129–137.

[16] J.B. Macqueen, Some methods of classification and analysis of multivariate observations, in: Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, 1967, pp. 281–297.

[17] X.L. Meng, D.B. Rubin, Maximum likelihood estimation via the ECM algorithm: A general framework, Biometrika 80 (2) (1993) 267–278.

[18] R. Neal, G.E. Hinton, A view of the EM algorithm that justifies incremental, sparse, and other variants, in: Learning in Graphical Models, 1998, pp. 355–368.

[19] C. Plant, C. Böhm, Parallel EM-clustering: Fast convergence by asynchronous model updates, in: 2010 IEEE International Conference on Data Mining Workshops, 2010, pp. 178–185.

[20] C.P. Robert, G. Casella, Monte Carlo Statistical Methods (Springer Texts in Statistics), Springer-Verlag New York, Inc., 2005.

[21] B. Thiesson, C. Meek, D. Heckerman, Accelerating EM for large databases, Mach. Learn. 45 (3) (2001) 279–299.

[22] UCI machine learning repository, http://archive.ics.uci.edu/ml/datasets.html.

[23] J. Wolfe, A. Haghighi, D. Klein, Fully distributed EM for very large datasets, in: ICML '08, 2008.

[24] J. Yin, L. Gao, Z.M. Zhang, Scalable nonnegative matrix factorization with block-wise updates, in: ECML/PKDD '14, 2014, pp. 337–352.

[25] J. Yin, L. Gao, Scalable distributed belief propagation with prioritized block updates, in: CIKM '14, 2014, pp. 1209–1218.

[26] J. Yin, L. Gao, Asynchronous distributed incremental computation on evolving graphs, in: ECML/PKDD '16, 2016, pp. 722–738.

[27] J. Yin, Y. Liao, M. Baldi, L. Gao, A. Nucci, GOM-Hadoop: A distributed framework for efficient analytics on ordered datasets, J. Parallel Distrib. Comput. 83 (C) (2015) 58–69.

[28] J. Yin, Y. Zhang, L. Gao, Accelerating expectation-maximization algorithms with frequent updates, in: CLUSTER '12, 2012, pp. 275–283.

[29] K. Zhai, J. Boyd-Graber, N. Asadi, M.L. Alkhouja, Mr. LDA: A flexible large scale topic modeling package using variational inference in MapReduce, in: WWW '12, 2012, pp. 879–888.

[30] Y. Zhang, Q. Gao, L. Gao, C. Wang, iMapReduce: A distributed computing framework for iterative computation, in: DataCloud '11, pp. 1112–1121.

[31] Y. Zhang, Q. Gao, L. Gao, C. Wang, PrIter: A distributed framework for prioritized iterative computations, in: SOCC '11, 2011, pp. 13:1–13:14.

[32] Y. Zhuang, H. Gao, F. Wu, S. Tang, Y. Zhang, Z. Zhang, Probabilistic word selection via topic modeling, IEEE Trans. Knowl. Data Eng. 27 (6) (2015) 1643–1655.

Jiangtao Yin received his Ph.D. degree in electrical and computer engineering from the University of Massachusetts at Amherst, in 2016. He is currently working at Palo Alto Networks, USA. His current research interests include data-intensive computing, large-scale data mining, and distributed systems.

Yanfeng Zhang received the Ph.D. degree in computer science from Northeastern University, China, in 2012. He is currently an associate professor at Northeastern University, China. His research consists of distributed systems and big data processing. He has published many papers in the above areas. His paper in ACM Cloud Computing 2011 was honored with "Paper of Distinction".

Lixin Gao is a professor of Electrical and Computer Engineering at the University of Massachusetts at Amherst. She received her Ph.D. degree in computer science from the University of Massachusetts at Amherst in 1996. Her research interests include social networks, and Internet routing, network virtualization and cloud computing. Between May 1999 and January 2000, she was a visiting researcher at AT&T Research Labs and DIMACS. She was an Alfred P. Sloan Fellow between 2003–2005 and received an NSF CAREER Award in 1999. She won the best paper award from IEEE INFOCOM 2010, and the test-of-time award in ACM SIGMETRICS 2010. Her paper in ACM Cloud Computing 2011 was honored with "Paper of Distinction". She received the Chancellor's Award for Outstanding Accomplishment in Research and Creative Activity in 2010. She is a Fellow of the ACM and a Fellow of the IEEE.