DISTRIM: Parallel GMM Learning on Multicore Cluster

Renyong Yang[1], Tengke Xiong[1], Tao Chen[2], Zhexue Huang[1], Shengzhong Feng[1]
[1]Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Shenzhen, China
Email: {ry.yang, tk.xiong, zx.huang, sz.feng}@siat.ac.cn
[2]EMC Labs China, China
Email: {tao.chen2}@emc.com

***Abstract*—Learning GMM model on extreme large data is challenging. We provide theoretical support for the feasibility of parallel EM-based GMM learning via distributed computing, and also design and implement a distributed memory sharing GMM learning system on multicore clusters, which is named as Distrim. Distrim aims to maximize the usage of computational power and minimize the communication overheads as much as possible. The experimental results show that Distrim is much more efficient than Hadoop, and also has a good scalability with respect to the number of computing nodes.**

Keywords: Gaussian Mixture Model, parallel learning, MPI, memory sharing, distributed computing

I. Introduction

Gaussian Mixture Model (GMM) [1] is a parametric probability density function represented as a weighted sum of Gaussian component densities, which has been widely used in a variety of applications such as speaker identification [2], [3] , image segmentation [4] and distributed data stream clustering [5]. There are several methods for estimating GMM parameters, such as moment estimation and maximum likelihood estimation (MLE). The MLE in GMM learning can be implemented through several machine learning methods, such as expectation-maximum (EM) [6], gradient descent [7], conjugate gradient [8], Gauss-Newton algorithms [9], etc. Among these methods, EM is the most widely used method for GMM learning due to its well-established theoretical framework.

However, EM in GMM learning is a computationally intensive task. First, EM is an iterative algorithm which requires many iterations to finish the learning procedure, K-means [10] is a case in point, which suffers a lower bound of $2^{\Omega(\sqrt{n})}$ (n is the number of observations) iterations for a complete learning process. EM even tends to involve far more than 100 iterations on most data sets [11]. Second, during each iteration, tremendous computation imposes on the performance evaluation of each observation, which is measured by log-likelihood function. Therefore, the classic serial EM learning running on a single machine is unable to handle the large data containing millions or even more observations. Learning from such large data has to resort to parallel approaches.

Concerning with the limitation of classic serial EM learning of GMM on large data, various parallel approaches have been proposed. Existing parallel approaches can be divided as MapReduce [12] approaches and non-MapReduce approaches. MapReduce is a popularly used parallel computational framework which has attracted many attentions in large data analysis. However, MapReduce does not directly support iterative learning process such as PageRank and EM in GMM learning [13]. Hadoop [14], developed based on MapReduce concept, is the most successful open-source platform. However, the iterations of GMM learning have to be performed by rounds of Hadoop jobs, which requires manually orchestrating execution using a driver program; furthermore, the creation of these jobs leads to tremendous redundant disk I/O, network traffic and time consuming. HaLoop [15] and Twister [16] have been proposed to be iterative-aware for MapReduce. The disk caching (not in memory) and indexing adopted in HaLoop accelerate the intermediate data accessing for application such as PageRank. However, these strategies expect little promotion in GMM learning because GMM learning process produces much little intermediate data. Twister is a full-in-memory iterative-aware framework, which caches all data in distributed memory pool; however, its dynamic scheduling pays considerable overheads for its runtime system, and data can not be shared among the daemon processes.

Some non-MapReduce parallel approaches have also been applied to GMM learning, such as the MPI approaches [17] and the memory-sharing approaches [18]. The MPI implementation [17] is designed for multiple single core clusters. Its intermediate computation needs many times of MPI *broadcast* operation, resulting in a none-optimized communication. The memory sharing implementation using CUDA running on GPU systems in [17], which is designed for parallel GMM learning, is the closest related to our work in this paper. In this implementation, each iteration of EM is split into six kernels, taking advantage of the high efficiency of GPU's SIMT architecture. Furthermore, the multithreaded data sharing scheme is able to save time on data exchange between processes and eliminate redundant data copies among processes. However, in its fifth kernel of the implementation, there is an intermediate data structure used for computing new covariance parameters, which grows with the increase of the size of input data, making it difficult to scale up for large data.

In this paper, we propose a new memory sharing parallel iterative-aware GMM learning on multicore clusters for large data. First, we provide theoretical support for the feasibility of using parallel partitioning and computation for EM based

978-1-4673-0089-6/12/$26.00 ©2012 IEEE

GMM learning. Our theorem shows that the learning process can be independent of the data size, which is superior for scaling up on large data. Second, concerning with the drawbacks of existing implementation for GMM learning (Hadoop, HaLoop, Twister, etc.) discussed above, we developed a new framework called *Distrim* from scratch, aiming to minimize space and communication overheads as much as possible, and to maximize the usage of computational power of multicore clusters as much as possible. Implemented with C++, Distrim has a distributed multithreaded architecture, which is scalable for multiple nodes, and can share data among threads within the same process and eliminate redundant data copies within processes. The experiments demonstrate that, compared with Hadoop implementation, Distrim gains notable improvement both in time and space consumption.

The rest of this paper is organized as follows: Section II introduces preliminary knowledge on GMM and its learning via EM algorithm. In section III, we provide theoretical support for parallel EM-based GMM learning via distributed computing. Section IV presents the framework of Distrim and its implementation for parallel GMM learning. In section V, experimental results are presented. We make conclusions in section VI.

II. Preliminary Knowledge on GMM Learning via EM

A GMM model is a weighted superposition of K components, each of which is modeled as multivariate Gaussian on continuous D-dimensional *row* vector $\mathbf{x} = \{x_1, \ldots, x_D\}$ with its own mean μ_k and covariance matrix $\mathbf{\Sigma}_k$ as parameters, $(1 \leq k \leq K)$. A GMM model is as follows:

$$P(\mathbf{x}|\theta) = \sum_{k=1}^{K} w_k \mathscr{N}(\mathbf{x}|\mu_k, \mathbf{\Sigma}_k)$$
$$\theta = \{w_1..w_K, \mu_1..\mu_K, \mathbf{\Sigma}_1..\mathbf{\Sigma}_K\} \tag{1}$$

where w_k is called a weight coefficient satisfying $\sum_{k=1}^{K} w_k = 1$. Each component λ_k of GMM is a D-variant Gaussian density function, which is as follows:

$$\mathscr{N}(\mathbf{x}|\mu_k, \mathbf{\Sigma}_k) = \frac{1}{(2\pi)^{D/2}|\mathbf{\Sigma}_k|^{1/2}} \exp\left\{-\frac{1}{2}(\mathbf{x}-\mu_k)\mathbf{\Sigma}_k^{-1}(\mathbf{x}-\mu_k)^{\mathrm{T}}\right\} \tag{2}$$

The EM algorithm for GMM learning is conducted through E and M steps. The E-step is to evaluate posterior ϕ_{nk}^t of observation $\mathbf{x}_n (1 \leq n \leq N)$ on the k-th component in the t-th iteration, i.e.,

$$\phi_{nk}^t = P(\mathbf{x}_n \in \lambda_k|\theta^{t-1}) = \frac{w_k^{t-1}P(\mathbf{x}_n|\theta^{t-1})}{\sum_{k=1}^{K} w_k^{t-1}P(\mathbf{x}_n|\theta^{t-1})} \tag{3}$$

Given these posteriors, the new estimates for the parameter set θ in the t-th iteration is obtained in the M-step, i.e.,

$$w_k^t = \frac{\sum_{n=1}^{N}\phi_{nk}^t}{N} \tag{4}$$

$$\mu_k^t = \frac{\sum_{n=1}^{N}\phi_{nk}^t \mathbf{x}_n}{\sum_{n=1}^{N}\phi_{nk}^t} \tag{5}$$

$$\mathbf{\Sigma}_k^t = \frac{\sum_{n=1}^{N}\phi_{nk}^t(\mathbf{x}_n-\mu_k^t)^{\mathrm{T}}(\mathbf{x}_n-\mu_k^t)}{\sum_{n=1}^{N}\phi_{nk}^t} \tag{6}$$

The *Evaluation* step uses log-likelihood function as convergence criterion,

$$\mathscr{L}^t = \mathscr{L}(X|\theta^t) = \sum_{n=1}^{N}\ln\left\{\sum_{k=1}^{K} w_k^t \mathscr{N}(\mathbf{x}_n|\mu_k^t, \mathbf{\Sigma}_k^t)\right\} \tag{7}$$

The iterations of E and M steps continue until the log-likelihood converged to a given threshold or maximum times of iteration reached.

III. Parallel GMM Learning via Distributed Computing

In this section we prove the feasibility of parallel GMM learning via distributed computing, and propose optimized computation strategies to speed up the learning process.

A. Feasibility of Parallel GMM Learning via Distributed Computing

Following the notations in Section II, let θ^t denote estimated parameters in the t-th iteration, $\Phi^t = \{\phi_{ij}^t\}(1 \leq i \leq N, 1 \leq j \leq K)$ denote the posterior (COEFFICIENT-FRACTION) matrix for the N observations on the K mixture components, $M^t = \{m_{ij}^t\} = \{\phi_{ij}^t\mathbf{x}_i\}(1 \leq i \leq N, 1 \leq j \leq K)$ denote the MEAN-FRACTION matrix, and $S^t = \{s_{ij}^t\} = \{\phi_{ij}^t\mathbf{x}_i^{\mathrm{T}}\mathbf{x}_i\}(1 \leq i \leq N, 1 \leq j \leq K)$ denote the COVARIANCE-FRACTION matrix, where s_{ij}^t itself is a matrix of order $(N \times N)$. X is split into B partitions, denoted as $X = (X_1, \ldots, X_B)^{\mathrm{T}}$, X_i represents observation set in the i-th partition, which is called a *block*. Each block X_i is split into T smaller *page*s, denoted as $X_i = (X_{i1}, \ldots, X_{iT})^{\mathrm{T}}$. Let $\Phi_{X_{ij}}^t$, $M_{X_{ij}}^t$ and $S_{X_{ij}}^t$ denote the sub-matrices of Φ^t, M^t and S^t for page X_{ij} in the t-th iteration.

Theorem 1: θ^t can be obtained by computing $\Phi_{X_{ij}}^t$, $M_{X_{ij}}^t$ and $S_{X_{ij}}^t$ in parallel.

Proof: Notice that $\Phi_{X_{ij}}^t$, $M_{X_{ij}}^t$ and $S_{X_{ij}}^t$ can be processed in parallel. The k-th $(1 \leq k \leq K)$ mixture component is obtained by summarizing the k-th column of every $\Phi_{X_{ij}}^t$ to get $\Phi_{X_{ij},k}^t$, then the k-th column of Φ^t can be obtained by

$$\Phi_k^t = \sum_{i=1}^{B}\sum_{j=1}^{T}\Phi_{X_{ij},k}^t = \sum_{X_i\in X}\sum_{X_{ij}\in X_i}\sum_{\mathbf{x}_n\in X_{ij}}\phi_{nk}^t$$

■

Likewise, M_k^t and S_k^t can be obtained by

$$M_k^t = \sum_{X_i\in X}\sum_{X_{ij}\in X_i}\sum_{\mathbf{x}_n\in X_{ij}}\phi_{nk}^t\mathbf{x}_n$$

$$S_k^t = \sum_{X_i \in X} \sum_{X_{ij} \in X_i} \sum_{\mathbf{x}_n \in X_{ij}} \phi_{nk}^t \mathbf{x}_n^{\mathrm{T}} \mathbf{x}_n$$

Then w_k^t, μ_k^t and $\mathbf{\Sigma}_k^t$ can be evaluated sequentially as

$$w_k^t = \frac{1}{N}\Phi_k,\ \mu_k^t = \frac{M_k^t}{\Phi_k^t}$$

and

$$\begin{aligned}\mathbf{\Sigma}_k^t &= \frac{\sum_{n=1}^N \phi_{nk}^t(\mathbf{x}_n - \mu_k^t)^{\mathrm{T}}(\mathbf{x}_n - \mu_k^t)}{\sum_{n=1}^N \phi_{nk}^t} \\ &= \frac{\sum_{n=1}^N (\phi_{nk}^t \mathbf{x}_n^{\mathrm{T}}\mathbf{x}_n - \phi_{nk}^t \mathbf{x}_n^{\mathrm{T}}\mu_k^t - \phi_{nk}^t \mu_k^{t,\mathrm{T}}\mathbf{x}_n)}{\sum_{n=1}^N \phi_{nk}^t} + \mu_k^{t,\mathrm{T}}\mu_k^t \\ &= \frac{S_k^t}{\Phi_k^t} - \mu_k^{t,\mathrm{T}}\mu_k^t\end{aligned}$$

Thus θ^t can be obtained by computing $\Phi_{X_{ij}}^t$, $M_{X_{ij}}^t$ and $S_{X_{ij}}^t$ in parallel. These sub-matrices are concurrently calculated on the pages using collateral threads.

Compared with the implementation of GMM learning in [18], our approach is scalable for large data. From above analysis we can see that $\Phi_{X_{ij}}^t$, $M_{X_{ij}}^t$ and $S_{X_{ij}}^t$ require $O(TBKD^2)$ space to store intermediate data, where each page requires only $O(KD^2)$ space, which is irrelevant to the size of input data. Whereas, in the implementation in [18], its fifth kernel needs $O(NKD)$ space, which grows with data size, making it difficult to scale up.

Since the implementation in [18] considers only special case where the Gaussian covariance matrices are diagonal, we do not make time comparison with it, as our implementation deals with general situations.

B. Optimized Computing

Optimizing the bottlenecks in the computing can remarkably improve the efficiency of the whole system. There are three bottlenecks can be optimized, including the computation for log-likelihood convergence arbitration, for posteriors ϕ_{nk} and for matrix multiplication $\mathbf{x}_n^{\mathrm{T}}\mathbf{x}_n$.

(1) Convergence arbitration: We propose a strategy, called *Delayed Convergence Arbitration* (DCA), to avoid redundant convergence computation. Let $\mathscr{L}^t$ denote the log-likelihood found in the t-th iteration, directly after θ^t has been obtained, $\mathscr{L}^t$ can be evaluated, and if $\mathscr{L}^t - \mathscr{L}^{t-1} < C$, the convergence is justified. DCA takes advantage of the association between calculation of log-likelihood and calculation of model update of GMM, and conducts the two types of computation synchronously. Let $\mathscr{L}^{t-1} = \text{Ł}(\theta^{t-1})$ denote the log-likelihood calculation, and $\theta^t = Q(\theta^{t-1})$ denote the model update calculation in the t-th iteration, assuming T is the number of iterations before convergence, then the last two synchronizing processes are denoted as follows:

$$\left\{\theta^T = Q(\theta^{T-1}) || \mathscr{L}^{T-1} = \text{Ł}(\theta^{T-2})\right\}$$

and

$$\left\{\theta^{T+1} = Q(\theta^{T}) || \mathscr{L}^{T} = \text{Ł}(\theta^{T-1})\right\}$$

. We select θ^{T+1} as the final result because $\text{Ł}(\theta^{t+1}) > \text{Ł}(\theta^t)$. DCA is named because the convergence arbitration of the T-th iteration is delayed to the $(T+1)$-th iteration. This process is called $L0$ level optimization.

(2) ϕ_{nk}: Let $\mathbf{\Sigma}_k^{-1} = \{\sigma_{ij}\}_k (1 \le i \le D, 1 \le j \le D)$, for $\forall\ \mathbf{x} \in X$, let $\Delta_k = \mathbf{x} - \mu_k = \{\delta_i\}_k (1 \le i \le D)$, then the calculation of $(\mathbf{x} - \mu_k)\mathbf{\Sigma}_k^{-1}(\mathbf{x} - \mu_k)^{\mathrm{T}}$ can be optimized as

$$\begin{aligned}&(\mathbf{x} - \mu_k)\mathbf{\Sigma}_k^{-1}(\mathbf{x} - \mu_k)^{\mathrm{T}} \\ &= \begin{pmatrix}\delta_1 \\ \dots \\ \delta_D\end{pmatrix}_k^{\mathrm{T}} \begin{pmatrix}\sigma_{11} & \dots & \sigma_{1D} \\ \dots & \dots & \dots \\ \sigma_{D1} & \dots & \sigma_{DD}\end{pmatrix}_k \begin{pmatrix}\delta_1 \\ \dots \\ \delta_D\end{pmatrix}_k \\ &= \left(\sum_{i=1}^D \sigma_{ii}\delta_i^2 + 2\left(\sum_{i=1}^D \sum_{j>i}^D \sigma_{ij}\delta_i\delta_j\right)\right)_k\end{aligned}$$

This optimized computing can reduce $O(D^2)$ addition operations. This process is called $L1$ level optimization.

(3) $\mathbf{x}_n^{\mathrm{T}}\mathbf{x}_n$: It is noticed that $\mathbf{x}_n^{\mathrm{T}}\mathbf{x}_n$ remains unchanged between iterations, so the intermediate results can be stored for latter use. A disadvantage is that the space complexity for high dimensional or large inputs is tremendous, because the size of intermediate data is D times of the original input. However, for a cluster containing many machines with considerable memory, it is still possible to apply it to large data. This process is called $L2$ level optimization.

IV. Distrim

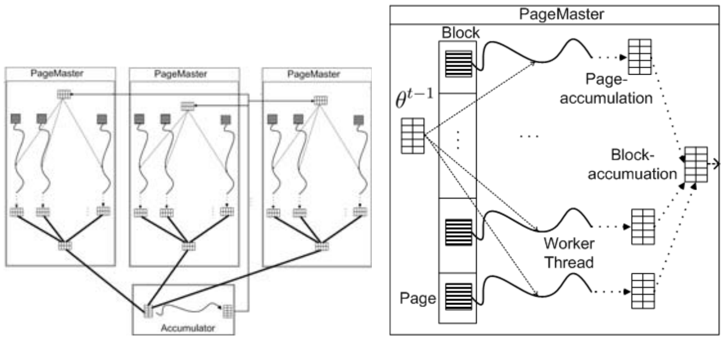
In this section, we describe the framework and detailed implementation of Distrim.

A. Framework of Distrim

The framework of Distrim aims to maximize the usage of computational power of multicore clusters, moreover, to minimize space and time consumption as much as possible. The design of the framework is divided into two parts, i.e., topology and scheduling.

The topology of Distrim is a static three-leveled hierarchical tree, as illustrated in Figure 1(a), where each leaf (in the third-level level) is a worker thread, and each second-level node is a *PageMaster* process, and its root is an *Accumulator* process. Assuming there are $(N+1)$ identically configured machines, Distrim assigns N machines N PageMaster processes, each machine for one process, and each process contains T collateral worker threads; the remaining one machine is used to load a single Accumulator process.

Conforming to aforementioned topology, Distrim adopts a statically orchestrated three-level hierarchical job scheduling and intermediate result accumulating. The first level of the scheduling is accomplished by worker threads within a PageMaster, each thread processing a page, in concurrence with others; and then the intermediate results are statistically scheduled to be accumulated to a page-accumulation within the same page. The second level of the scheduling is accomplished by a local thread, who is scheduled to accumulate all page-accumulations to a block-accumulation. The third level



(a) Architecture of Distrim (b) PageMaster

Fig. 1. Framework of Distrim

of the scheduling is accomplished by the Accumulator process, where all the block-accumulations are scheduled to be summarized to complete results via network. In this scheduling, the computing in each node is independent of that in other nodes in the same level, thus maximized parallel computing and minimized communication overhead are expected.

Distrim assumes all the input data is fit in-memory, which is suitable for the computational intensive GMM learning. The architecture of the PageMaster is illustrated in Figure 1(b), which consists of four entities, i.e., a set of in-memory observation pages, a set of worker threads and a local thread, a single copy of parameters and a set of intermediate data.

Our memory sharing framework has three advantages: first, the three-level accumulating greatly reduces network traffic by locally pre-accumulating the intermediate results within each PageMaster; second, the static scheduling eliminates time overheads compared with dynamic scheduling, which plays an important role in the enormous iterations; third, the parameters are shared among collateral threads, which eliminates space consumption resulting from storing redundant copies of identical parameters of multiple processes. So both time and space consumptions are optimized in Distrim.

Algorithm 1 Parallel GMM Learning on Distrim

```
Input: Training observation set $X$, maximum iterations $M$, convergence threshold $C$, initial restarts $R$, initial steps $S$, components $K$, number of computing nodes $W$.
Output: Learned parameter set $P$.
 1: Split $X$ into $X_1 \sim X_W$ and distribute $X_i$ to computing node $i$, $t \leftarrow 1$;
 2: for each restart $r (1 \leq r \leq R)$
 3:     $P_r^0 \leftarrow RunInitProc(r, K)$;
 4: for each restart $r$
 5:     Perform $S$ iterations for $r$: $P_r^t \leftarrow RunProc(P_r^{t-1})$;
 6: Add all restarts to restart set $Q$, $m \leftarrow R$, $n \leftarrow 1$;
 7: while $(m > 1)$
 8:     for each remaining restart $r$ in $Q$
 9:         Perform $n$ iterations for $r$: $P_r^t \leftarrow RunProc(P_r^{t-1})$;
10:     Exclude $m$ restarts with smaller log-likelihoods from $Q$. $m \leftarrow m/2$, $n \leftarrow n \times 2$;
11: while $(t < M$ and $\mathscr{L}_q^{t-1} - \mathscr{L}_q^{t-1} > C)$ /*$q$ is the only restart left in $Q$.*/
12:     $P_q^t \leftarrow RunProc(P_q^{t-1})$;
13: return $P_q^t$
```

B. Implementation Details

The pseudocode of Distrim implementation is illustrated in Algorithm 1 and Algorithm 2. Algorithm 1 launches multiple restarts at one time and terminates unnecessary iterations in the early stages, and then the best restart with the best final estimated parameters is selected as the final result.

There are two main procedures in Algorithm 1, i.e., *RunInitProc* and *RunProc*. The *RunInitProc* procedure randomly initializes the parameters for multiple restarts, and randomly assigns each observation to one of the mixture components, then estimates the initial parameters using the observations within the component.

The inner iterative-aware procedure, i.e., *RunProc*, is the core of our implementation. *RunProc* works as follows: worker threads $\mathrm{Thrd}_i (1 \leq i \leq T)$ run in parallel to process each page within each PageMaster, then the intermediate accumulations within each page are accumulated to the block-accumulation, which are then be accumulated by the Accumulator process for computing the new set of parameters of the next iteration, as illustrated in Algorithm 2.

V. Experimental Results

In this section, we present the experimental results of Distrim and compare the performance of Distrim with that of Hadoop.

A. Environment

Instead of programming in Java, Distrim is implemented with C++, as C++ can control memory allocation and deallocation completely. MPI is adopted as the underlying runtime for Distrim, as it has low communication overhead thanks to its direct transmission on TCP/IP. The well-established library Boost[1] is adopted to manage threads.

The experiments are conducted on a cluster consisting of

[1]http://www.boost.org

Algorithm 2 Multithreaded $P^t \leftarrow RunProc(P^{t-1})$ Procedure

```
Input: Training observation subset $X_i$ in the $i$-th computing node, old parameter estimate $P^{t-1}$, number of computing nodes $W$.
Output: New parameter estimate $P^t$.
1:  In $i$-th $\mathrm{PageMaster}_i$:
2:    $T \leftarrow GetCoreNumber()$, $\mathrm{Thrd_{init}}$ created to allocate $T$ pages, scatter $X_i$ into pages, worker threads $\mathrm{Thrd}_j (1 \leq j \leq T)$ and $\mathrm{Thrd}_0$, sleep;
3:    repeat (RunProc):
4:      $\mathrm{Thrd}_0$ sleep;
5:      $\mathrm{Thrd}_i (1 \leq i \leq T)$: fetches a page, processes its observations with $P^{t-1}$, sleep; $\mathrm{Thrd}_q$ wakes up $\mathrm{Thrd}_0$, sleep;/* $\mathrm{Thrd}_q$ is the last active thread. */
6:      $\mathrm{Thrd}_0$: $I_i \leftarrow I + I_{\mathrm{Thrd}_i} (1 \leq i \leq T)$, send $I_i$ to Accumulator, awaiting $P^t$ from Accumulator;
7:      $\mathrm{Thrd}_0$: receives $P^t$, $P^{t-1} \leftarrow P^t$, wakes up $\mathrm{Thrd}_1 \sim \mathrm{Thrd}_T$;
8:    until end
9:    $\mathrm{Thrd}_0$ and $\mathrm{Thrd_{init}}$ awake and clean up resources, then exit;
10: Accumulator (RunProc):
11:   while (!AllRestartsConverged())
12:     $I \leftarrow I + I_i$, compute $P^t$ from $I$, broadcast $P^t$ to $\mathrm{PageMaster}_i$ $(1 \leq i \leq W)$;
```

30 identically configured nodes, each node has 15GB memory and two Intel Xeon CPUs, and each CPU contains eight 2.4Gz cores. A 1,000Mb Ethernet switch is used to connect all the nodes. Each node is installed with a 64bit CentOS 5.5 Linux System. Hadoop is v0.20.2 with official JDK 1.6.0_21. OpenMPI [19] v1.4.3 is used in the MPI runtime system, and Boost is v1.46.2.

B. Datasets

Being constrained by the hardware configuration, the size of input data is limited to 100 million observations with 10 dimensions (10GB), or equally 10 million observations with 100 dimensions. The synthetic data used in our experiment are generated by a data generator, which authentically generates data following the multivariate Gaussian distribution with given parameters. We keep the number of mixture components as FOUR in the data generation and model learning. There is no missing value in the datasets, and each observation is a text line containing double type values in each dimension, separated by commas.

C. Distrim vs. Hadoop

The comparison of Distrim with Hadoop is given in Figure 2. The size of input data for the left figure is 1GB, and the right is 10GB. On the 1GB data, Distrim was implemented with three versions, i.e., L0, L1 and L2 level optimization. On the 10 GB data, Distrim was implemented with two versions, i.e., L0 and L1 level; L2 level optimization is not included because it can not be applied to our cluster due to the memory limitation. On each data set, Distrim and Hadoop is compared by iteration time on three different dimensionalities, i.e., 10, 50 and 100 dimenstions.

From Figure 2, we can see that Distrim always outperforms Hadoop. If the intermediate data can fit into the memory, i.e., L2 can be applied, Distrim outperforms Hadoop even better. We also observe that, dealing with 1GB scale data only needs 1.38 second in the best case, while Hadoop needs 85.33 seconds, speeding up to 61.8 times. Furthermore, we can see that L1 level optimization gains small improvement over L0 level optimization, while L2 level optimization speeds up the GMM learning by a large margin from L1 level.

D. Scalability w.r.t Number of Nodes

In this part, we test the scalability of Distrim with respect to number of nodes, and compare it with Hadoop. The scalability w.r.t number of nodes is also tested on two different data sets, i.e., 1GB and 10GB. On each data set, we test on three different dimensionalities, i.e., 10, 50 and 100 dimensions. The number of computing nodes varies from 5 to 29, and Accumulator process runs on an extra node.

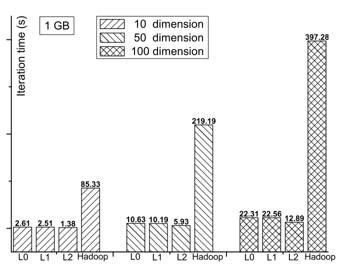
The comparison of Distrim with Hadoop is illustrated in Figure 3. From Figure 3 we can see that Distrim always outperforms Hadoop in terms of scalability w.r.t the number of nodes. It is surprising seeing that, the iteration time of Hadoop on the 1GB data set (Figure 3(a), 3(c) and 3(e)) fluctuates around a mean value. Adding more computing nodes does not help decreasing the iteration time, even leading to an increase of iteration time. Also, from Figure 3(b) and Figure 3(d), we can see a reverse of iteration time when the number of nodes increases from 25 to 29. The reason is twofold: first, when more nodes are added to the cluster, the complex runtime system of Hadoop has to consume more time on maintaining the tracking and scheduling tasks; second, an imbalanced data replication may lead to an imbalanced data processing.

VI. Conclusions

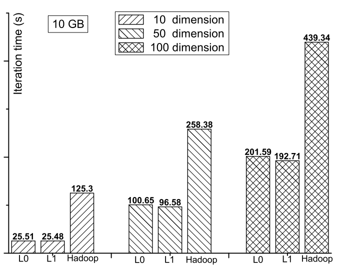
In this paper, we have provided theoretical support for the feasibility of parallel EM based GMM learning via distributed computing, and proposed three optimization strategies to improve the efficiency of learning process. Based on the theoretical analysis and optimization strategies, we have proposed and implemented a comprehensively optimized parallel learning system on multicore clusters for GMM model, named Distrim. Our system promotes memory sharing among threads, maximizes the usage of computational resources of multicore cluster and decreases time and space consumption. The experimental results demonstrated that Distrim outperforms Hadoop by a large margin in terms of both efficiency and scalability w.r.t the number of computing nodes.

Acknowledgements

This work is supported by Shenzhen Key Laboratory of High Performance Data Mining(grant no. CXB201005250021A).

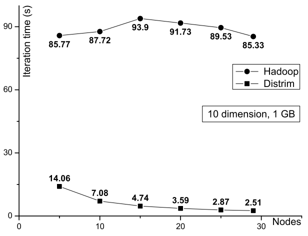


(a) 1GB

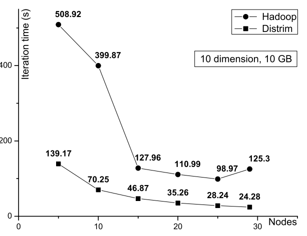


(b) 10GB

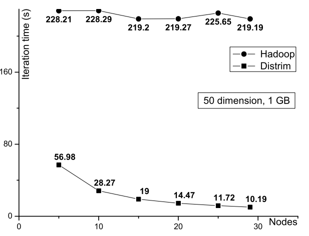
Fig. 2. Iteration time comparison between Distrim and Hadoop



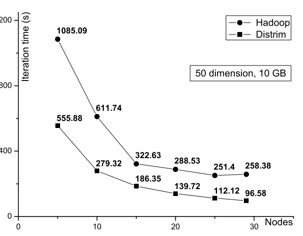
(a) 10 dimension, 1GB



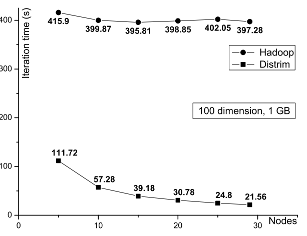
(b) 10 dimension, 10GB



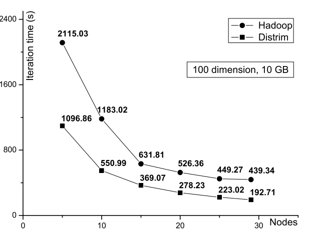
(c) 50 dimension, 1GB



(d) 50 dimension, 10GB



(e) 100 dimension, 1GB



(f) 100 dimension, 10GB

Fig. 3. Scalability Comparison between Hadoop and Distrim

References

[1] G. J. Mclachlan and K. E. Basford. Mixture models: inference and applications to clustering. 1988.

[2] Douglas A. Reynolds and Richard C. Rose. Robust text-independent speaker identification using gaussian mixture speaker models. *IEEE Transactions on Speech and Audio Processing*, 3:1, January 1995.

[3] L. Rabiner and B.H. Juang. *Fundamentals of Speach Recognition*. Prentice Hall Signal Processing Series, 1993.

[4] Rahman Farnoosh and Behnam Zarpak. Image segmentation using gaussian mixture model. *IUST International Journal of Engineering Science*, 19:29–32, 2008.

[5] Aoying Zhou, Feng Cao, Ying Yang, Chaofeng Sha, and Xiaofeng He. Distributed data stream clustering: A fast em-based approach. In *IEEE 23rd International Conference on Data Engineering*, April 2007.

[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 (Methodological)*, 39(1):1–38, 1977.

[7] K. Lange. A gradient algorithm locally equivalent to the em algo-rithm. 1995.

[8] Hestenes Magnus R. and Stiefel Eduard. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 1952.

[9] Björck. Numerical methods for least squares problems. *Society for Industrial and Applied Mathematics*, 1996.

[10] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. *Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability*, 1, 1967.

[11] Claudia Plant and Christian Bhm. Parallel em-clustering: Fast convergence by asynchronous model updates. *IEEE Internet Computing*, pages 178–185, 2010.

[12] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, California, USA, December 6-8 2004. USENIX Association.

[13] Cheng tao Chu, Sang Kyun Kim, Yi an Lin, Yuanyuan Yu, Gary Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. In *Advances in Neural Information Processing Systems 19*, pages 281–288, 2007.

[14] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, 2009.

[15] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael Ernst. Haloop: Efficient iterative data processing on large clusters. *Proceedings of The Vldb Endowment*, 3:285–296, 2010.

[16] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In *IEEE International Symposium on High Performance Distributed Computing*, pages 810–818, 2010.

[17] Pedro E. Lpez de teruel, Jos M. Garca, and Manuel E. Acacio. The parallel em algorithm and its applications in computer vision. In *Parallel and Distributed Processing Techniques and Applications*, pages 571–578, 1999.

[18] N. S. L. Phani Kumar, Sanjiv Satoor, and Ian Buck. Fast parallel expectation maximization for gaussian mixture models on gpus using cuda. In *High Performance Computing and Communications*, pages 103–109, 2009.

[19] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. *Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation*. 2004.