Parallel Implementation of Expectation-Maximization for Fast Convergence

Henggang Cui
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213
hengganc@ece.cmu.edu

Jinliang Wei
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
jinlianw@cs.cmu.edu

Wei Dai
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
wdai@cs.cmu.edu

1 Introduction

Latent variable models have become popular tools for modeling data that potentially arise from latent causes. This class of models includes the unstructured Gaussian mixture model (GMM), temporally linked Hidden Markov Model (HMM), and the more complex latent Dirichlet allocation (LDA) [3]. Despite the varying latent variable structures, these models all poses challenging inference problem as the latent variables are not directly observed. Solution techniques are generally only approximate, such as the Expectation-Maximization (EM) algorithm and sampling-based methods such as Gibbs sampling. In most cases, these approaches are computationally intensive. As the data set today becomes ever more gigantic, there is a pressing need for efficiently parallelizing these inference algorithms in multi-core and distributed environment.

A large body of work has focused on parallelizing the sampling-based algorithms [1, 14]. These solutions are generally designed for particular models and requires much engineering efforts. On the other hand, in recent years there has been an explosion of novel parallel frameworks designed for machine learning tasks [11, 8, 15, 13]. These new frameworks recognize that traditional parallel schemes such as the messge passing interface (MPI) or Map-Reduce are insufficiently expressive or inefficient for machine learning algorithms, and various system design and programming paradigms are proposed [11, 15, 13]. These frameworks depart from the disk-based computation used in frameworks like MapReduce and instead use in-memory execution for fast iteration. However, the relatively ad-hoc choices of parallel abstractions and implementations may be effective for a particular set of ML techniques, but to the best of our knowledge there is no comprehensive survey of these frameworks in terms of performance and programming paradigm. Furthermore, EM algorithm, a fundamental building block of many sophisticated techniques [9], is largely overlooked within the scalable machine learning community, potentially due to the unique challenge in parallelization strategy. In this work we investigate the parallel strategy and performance for EM algorithms on GMM using three distinctive frameworks: GraphLab [12, 7], Piccolo [13, 4], and Spark [15]. Here we briefly summarize the features of each framework and their respective parallel strategies.

- **GraphLab**: GraphLab abstracts data and computation by a directed graph. It let users assign data and corresponding computation with vertices and edges in the graph. The vertex-program can be executed in parallel on each vertex and can interact with neighboring vertices. In contrast to the more general message passing and actor models, GraphLab constrains the interaction of vertex-programs to the graph structure, which enables a wide range of system optimizations. When each vertex gathers the data of its neighboring edges and vertices, it's actually like doing a local map-reduce computation. With the graphic structure, GraphLab makes multiple map-reduce compuations happen concurrently. GraphLab is built on top of MPI and C++ boost library, and programmers implement GraphLab program through the provided C++ API.

- **Piccolo**: Piccolo uses *distributed key-value store* (hash table) partitioned on to the worker nodes according to user-specified policy. The flexibility of hash table offers the most general programming abstraction among the three. However, such flexbility exposes low level system details (e.g. the table partition policy) that could create unnecessary burden on the programmers. The execution model is synchronous as it imposes global barrier at the end of each (synchronous) step. A later version (Oolong) also support asynchronous execution through table "trigger." Users program in C++ or, with a more restricted support, Python. Fault tolerance is achieved through asynchronous checkpointing.
- **Spark**: Spark offers efficient fault tolerance by using a restricted abstraction of distributed shared memory: the Resilient Distributed Datasets (RDDs). An RDD is a read-only, partitioned collection of records. Users assemble data into RDDs and invoke Spark operations on each RDD to do computation. RDDs are partitioned to be distributed across nodes. *Transformation* operations on multiple partitions are run in parallel by Spark. The read-only feature of RDD enables efficient fault tolerance. Instead of checkpointing. Spark only needs to record the lineage of an RDD. Since RDD is read-only, updating one record in an RDD requires generating a new RDD, which is very inefficient. Therefore, RDDs are best suited for applications that apply the same operation to all elements of a dataset. This feature makes a Spark an eligible platform for EM algorithms, which usually repeatedly apply the same operations on a large set of data and aggregate the result.

2 Related Work

Parallel EM algorithm on mixture models is implemented in message passing paradigm in the context of computer vision [6]. [5] implements EM algorithm for a transformed version of probabilistic latent semantic models (PLSI) in MapReduce framework. Parallel EM algorithm was also implemented for HMM. The standard EM algorithm for HMM is Baum-Welch algorithm which consists of two sequential traversal of the hidden states [2] in its expectation (E) step. [10] proposes a cut-and-paste algorithm which cuts the sequence into multiple segments connected by an extra variable in the hidden Markov chain. Each processor execute the E step concurrently and combine the results at the paste stage. The algorithm is designed for learning linear dynamical systems (LDS). but can be extended to HMM. The implementation in OpenMP, a shared memory parallel library, demonstrates near linear speedup for small number of processors.

3 Method

3.1 Gaussian Mixture Model

We have implemented EM algorithm on Gaussian Mixture Model (GMM). We define N as the number of data points, K as the number of gaussian mixtures, and P as the dimension of each data point. For GMM, the corresponding E-step and M-step is as follows.

1. The E-step estimates the posterior probabilities using the current model parameters.

$$\gamma(z_{nk}) = \frac{\pi_k N(x_n|\mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j N(x_n|\mu_j, \Sigma_j)} \tag{1}$$

2. The M-step then maximizes the log likelihood w.r.t. the mixing coefficients (π_k), gaussian means (μ_k), and gaussian covariances (Σ_k).

$$N_k = \sum_{n=1}^{N} \gamma(z_{nk}) \tag{2}$$

$$\mu_k^{new} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk}) x_n \tag{3}$$

$$\Sigma_k^{new} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk})(x_n - \mu_k^{new})(x_n - \mu_k^{new})^T \tag{4}$$

$$\pi_k^{new} = \frac{N_k}{N} \tag{5}$$

In our application, we assume that the number of gaussian mixtures (K) and the dimension (P) are small. But the number of data points (N) is very large (could be millions), so that it's the major performance bottle neck. In the implementation, we should make the computation traversing all the data points $1 \sim N$ parallel.

3.2 Single Machine Implementation

We first implemented a single threaded EM algorithm in C++, which is reused in GraphLab and Piccolo's implementations. Our implementation follows the approach of a standard Matlab implementation[1]. This implementation is robust against numerical underflows even when Gaussian probability is exponentially small. In our C++ implementation we take a further optimization step to precompute the leading coefficient $(2\pi)^{-d/2}|\Sigma|^{-1/2}$ and Σ^{-1} before each E-step. We found the run time of our C++ implementation is within 5

3.2.1 Parallel GMM Implementation on GraphLab

Here we discuss the detailed parallel implementation of EM algorithm (on GMM model) on GraphLab. Since we assume the number of data points is the major performance bottle neck, we assign each data point (x_n) along with its latent variable (z_n) to a vertex in the graph. Because the number of gaussian mixtures (K) and the dimension (P) are small, we store the mixing coefficients (π_k), gaussian means (μ_k), and gaussian covariances (Σ_k) as global variables. These global variables are read-only inside the vertex program and will only be updated centrally.

For the E-step, the computation is completely data parallel, in that the computation of $\gamma(z_n)$ only depends on the value of its corresponding data point x_n and the current model parameters. GraphLab offers a method called transform_vertices(), which would invoke a certain computation on all the vertices. We use this method to implement E-step.

For the M-step, the computation has two part. Firstly, each vertex should calculate its contribution to the model paramter update. Then the values should be summed up across all the vertices. This computational pattern is very similar to MapReduce. If we interpret this process in MapReduce language, the Map function would be

$$Vertex_n \rightarrow \{(\text{“}N_k\text{”}, \gamma(z_{nk})), (\text{“}\mu_k\text{”}, \gamma(z_{nk})x_n)\}$$

The Reduce phase should just sum up the values associates with each key.

GraphLab offers a mechanism called map_reduce_vertices() that allows user to do MapReduce computing. Since the calculation of Σ_k needs the updated value of μ_k, we use two rounds of MapReduce calculations to implement M-step. The first round calculates N_k, π_k, and μ_k. And the second round calculates Σ_k. [2] Finally, these values will be updated to the global model parameter centrally, where we also check the convergence condition and possibly run E-step agian.

One thing about this implementation is that it's actually not the typical way of how GraphLab works, because in the corresponding graph, there's no edges. Alternatively, we can store each Gaussian mixture component in a vertex instead of storing them as global variables. But if we do so, the vertices containing the Gaussian mixture components will have to be connected to all the vertices containing data, and the graph will be very dense. Since the GraphLab toolkit package implements

[1] http://www.mathworks.com/matlabcentral/fileexchange/26184-em-algorithm-for-gaussian-mixture-model

[2] Actually after we finished the implementation, we realize that we can calculate Σ_k from $\sum_{n=1}^{N} \gamma(z_{nk})x_n x_n^T$ and μ_k, so that we can do the whole M-step in one MapReduce round. But we didn't change our implementation due to time pressure.

K-means algorithm by keeping the current K clusters as global variables (which is exactly the same approach as ours), we believe that is more efficient than having a dense graph.

3.3 Parallel GMM Implementation on Piccolo

Piccolo achieve parallelization through global, distributed hash table. The following tables are created to efficiently keep track of the program state:

- **Responsibility**: A $N \times K$ table storing the log-likelihood for each point, which is estimated in E-step (log-likelihood is used for numerical stability). Each shard of the table contains $1/W$ points where W is the number of worker nodes. The accumulator associated with each table entry is `Replace` as each E-step recomputes the responsibility of each data point.
- **Sum of responsibilities**: A $1 \times K$ table to store N_k in Eqn. 2. This table is continuously updated during E-step to avoid an additional global step to sweep through the Responsibility table. The associated accumulator is `Sum` so that continuous update to the entry during one E-step will be accumulated property. This table needs to be zero at the end of M-step.
- **Model parameters**: Stores GMM model parameters, including the mixture weights, the Gaussian component means, the component covariance Σ_k and its determininant and inverse. The associated accumulator is `Sum` so that all workers can stream partial updates to the table.

Since communication over network incurs much higher latency than reading values stored in local memory, we cache the model parameters in E-step so that only one remote read of all model parameters is necessary. Note that this is a blocking remote read, as no responsibility could be computed without the model parameters from last M-step. There is still remote write to *sum of responsibilities* table, but it's buffered and non-blocking. In M-step each worker compute partial parameter updates in parallel and make buffered remote write. All tables are evenly sharded and distributed, including the smaller tables, for reduced network latency.

3.4 Parallel GMM Implementation on Spark

As discussed above, in Spark, users assemble data into RDDs and invoke Spark operations on each RDD to do computation. RDDs can only be created through reading data from stable stroage or Spark operations over existing RDDs. Spark defines two kinds of RDD operations: 1) *transformation*s, which apply the same operation[3] on every record of the RDD to generate a separate, new RDD; 2) *action*s, which aggregate the RDD to generate computation result. Note that in *transformation*, an operation on a record is independent from other records in the RDD.

Each RDD is partitioned. Partitions of an RDD are distributed across multiple nodes and *transformation*s are run in parallel on all partitions. However, since *action*s are aggregating one RDD, it can only be run sequentially, iterating through every record in the RDD. For efficiency, Spark requires the *reduce* function of *action*s to be commutative so that Spark can run *reduce* on parallel on each node and aggregate the results from each node.

As we have discussed, RDDs are best suited for applications that apply the same operation to all elements of a dataset. In Gaussian Mixture Model, for each E-step, we calculate the conditional probabilty density of each Gaussian distribution at every point x_i, and for each M-step, we aggregate the result from E-step to update each Gaussian models and priors of all models.

Therefore, naturally, each E-step is a Spark *map* transformation which runs in parallel mapping each x_i to a vector of conditional probability densities, and each M-step is a *reduce* action which goes through all the records in the RDD, aggregating results from E-step. In our GMM implementation, each iteration consists of two map operations and two reduce operations. In the first map operation, we calculate the resposibility (γ) of each cluster for each data point, along with the product of the data point and γ and the sum of the products for all clusters. Then we do a reduce operation to

[3]Spark is implemented in Scala. Each function is also an object. A mapping function that maps a record in the RDD to a new record is supplied to a *transformation* operation as a paramter.

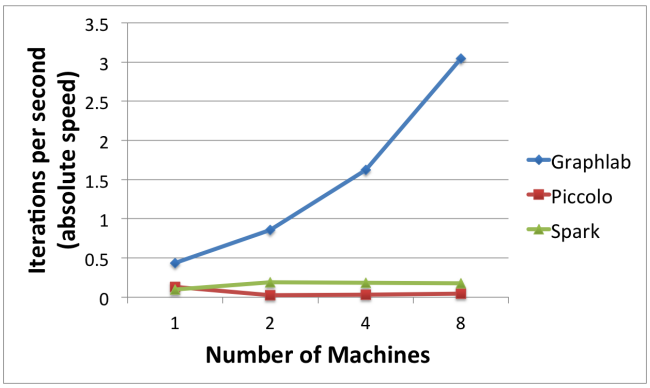

Figure 1: Comparison of absolute speed: of iterations per second with fixed problem size

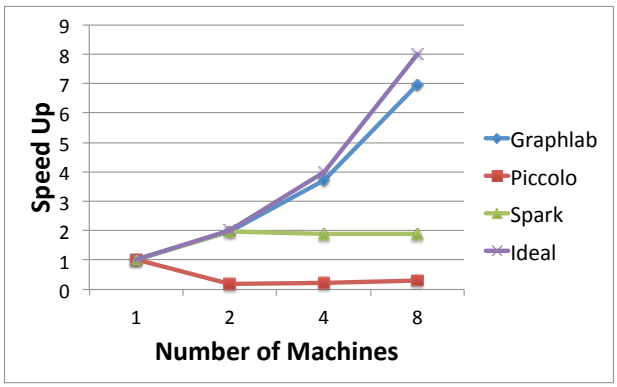

Figure 2: Strong scaling: speedup vs. number of machines

calculate the new centers for each cluster. In the las step, we do another map and reduce to calculate the covariance of each cluster.

4 Experimental results

For performance evaluation of the three franeworks, we focused on two aspects: scalability and absolute speed. We run all three frameworks on virtual machines running on CMU PDL lab's OpenCirrus cluster. Each virtual machine has 1 processor (AMD Opteron 244) and 2GB RAM memory, and runs Linux debian 3.2.0. In order to test scalability, we vary our cluster size from 1 to 8 VMs. [4]

We used synthetized data set. The number of gaussian distributions (K) is 4, and the dimenstion of data point (P) is 2. Size of the data set ranges from 1×10^6 to 8×10^6.

4.1 Comparison of absolute speed

Fig. 1 shows the comparison of abosolute speed of the three frameworks. The graph shows that GraphLab is $2x$ to over $10x$ faster than Spark and Piccolo. Piccolo suffers much heavier communcation cost. In Spark, the algorithm spends significant amount of time (around 90%) on the two sequential summation operation. Also, even tough Scala has comparable performance to C++ in most case, Scala is much less efficient for serialization and deserialization.

[4] Piccolo and Spark use 1 additional machines as the master node.

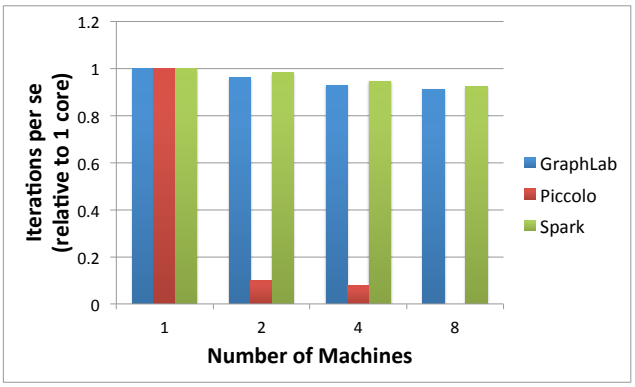

Figure 3: Weak scaling: of iterations per seccond with fixed data size per machine

4.2 Strong scaling

We evaluate the strong scaling of our implementation by fixing the number of data points to be 1 million, and run the calculation on 1, 2, 4, 8 working machines. Fig. 2 shows the finishing time. For GraphLab, the speed up is almost linear in the number of machines. However, Piccolo suffers from the huge communication cost when running on more than one machine, thus there is negative speed up from 1 to 2 machines. This is despite the fact that we have extended the Piccolo framework to support local cache control to reduce number of remote read. From two machines, Piccolo does scale up modestly. Changing from 1 machine to 2, Spark gets nearly $2x$ speedup. However, further increase of the number of machines does not result in higher speedup. We suspect it is most likely because of the dominant sequential steps.

4.3 Weak scaling

We also evaluate the weak scaling by fixing the number of data points per core. For cluster with 1, 2, 4, 8 machines, we run experiments on 1, 2, 4, 8 million data points. We measure iterations per second as a proxy for total throughput (Fig. 3). Note that different data set may need different number of iterations to converge. The result shows that our implementations on GraphLab and Spark exhibits excellent weak scaling. But we notice that as the number of machines increases, communication overhead results in some small decrease on the throughput.

For Piccolo, again the communication dominates computation and killed the performance. It degrades rapidly from single core to two cores, but only modestly from two cores to four cores. We were not able to complete the computation on 8 cores due to the large data size overburdening the underlying MPI communication.

5 Discussion

In Piccolo's implementation, E-step is the computation intensive step, wherease the M-step is mainly bottlenecked by communication. It is interesting to observe that the run time breaks down about equally between E-step and M-step for Piccolo. This fact further underlies that the performance of parallel frameworks is highly dependent on both the machine runtime and the communication latency.

Piccolo's run time on single core is strangely faster than multi-core because Piccolo uses a very poor serialization method. Since GMM has relatively large parameter variables, communication costs dominate the computation. Even for single core, both Piccolo and Spark is about 3 times slower than GraphLab, demonstrating the excellent framework building in GraphLab.

Spark uses similar computation model to MapReduce/Hadoop. Even though Spark achives much better perforamce than Hadoop by carefully caching data in memory, this feature is not uitilzed in our evaluation since the dataset well fits in the machines' memory. Like Hadoop, the efficiency of

	Lines of code
GraphLab	489
Piccolo	637
Spark	234

Table 1: Number of lines of user code for each implementation of EM-GMM on GraphLab, Piccolo, and Spark.

Spark highly depends on the parallelism of the algorithm itself. When there is a costly sequential step, Spark performs poorly.

To gauge usability and programmer productivity of each framework, we tally the size of user code in Table 1. The numbers exclude empty lines and the codes are minimally commented. All three implementations use external linear algebra libraries, which uses high level mathematical expressions and operations akin to Matlab. Spark achieves the most economical user code largely because the framework is based on Scala, which is a high level functional programming language. In contrast, both GraphLab and Piccolo uses C++ API's, and a significant portion of the user code was for low-level tasks such as serialization (marsharling) and deserialization (unmarshaling). We further point out that GraphLab and Piccolo derive from the same single-threaded EM algorithm implementation, so the difference between the two is mostly due to framework API.

6 Conclusion

Our EM implementation on GraphLab exhibits excellent strong and weak scaling for up to 8 machines. Good performance is partially due to the lack of latent variable structure, which makes the computation highly data parallel.

In terms of programming abstraction, EM algorithm for GMM is natural for Piccolo and Spark, but the "graph-like" representation in GraphLab would produce dense graph and incur much overhead. Thus we use the MapReduce implementation instead. Finally, we also found that all three frameworks have very poor support for basic mathematical operations, such as the ones needed to compute multi-variate Gaussian. We had to resort to external libraries, which often caused compatibility issues at both compile time and run time. Users of these frameworks will greatly benefit from basic linear algebra libraries natively supported in these frameworks.

Acknowledgments

We thank Parallel Data Lab (PDL) for providing computer cluster for our experiments. We also appreciate Greg Ganger, Qirong Ho, and Eric Xing's inputs to our work.

References

[1] Amr Ahmed, Mohamed Aly, Joseph Gonzalez, Shravan Narayanamurthy, and Alexander J. Smola. Scalable inference in latent variable models. In *WSDM*, pages 123–132, 2012.

[2] Leonard E. Baum, Ted Petrie, George Soules, and Norman Weiss. A Maximization Technique Occurring in the Statistical Analysis of Probabilistic Functions of Markov Chains. *The Annals of Mathematical Statistics*, 41(1):164–171, 1970.

[3] David M. Blei, Andrew Ng, and Michael Jordan. Latent dirichlet allocation. *JMLR*, 3:993–1022, 2003.

[4] Jinyang Li Christopher Mitchell, Russell Power. Oolong: Programming asynchronous distributed applications with triggers. SOSP, 2011.

[5] Abhinandan S. Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram. Google news personalization: scalable online collaborative filtering. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 271–280, New York, NY, USA, 2007. ACM.

[6] Pedro E. López de Teruel, José M. García, and Manuel E. Acacio. The parallel em algorithm and its applications in computer vision. In *PDPTA*, pages 571–578, 1999.

[7] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, Hollywood, October 2012.

[8] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[9] Qixia Jiang, Jun Zhu, Maosong Sun, and Eric P. Xing. Monte carlo methods for maximum margin supervised topic models. In *NIPS*, 2012.

[10] Lei Li, Wenjie Fu, Fan Guo, Todd C. Mowry, and Christos Faloutsos. Cut-and-stitch: efficient parallel learning of linear dynamical systems on smps. In *KDD '08: Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 471–479, New York, NY, USA, 2008. ACM.

[11] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Catalina Island, California, July 2010.

[12] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *PVLDB*, 2012.

[13] Russell Power and Jinyang Li. Piccolo: building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–14, Berkeley, CA, USA, 2010. USENIX Association.

[14] Yi Wang, Hongjie Bai, Matt Stanton, Wen-Yen Chen, and Edward Y. Chang. Plda: Parallel latent dirichlet allocation for large-scale applications. In *Proceedings of the 5th International Conference on Algorithmic Aspects in Information and Management*, AAIM '09, pages 301–314, Berlin, Heidelberg, 2009. Springer-Verlag.

[15] Matei Zaharia, N. M. Mosharaf Chowdhury, Michael Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. Technical Report UCB/EECS-2010-53, EECS Department, University of California, Berkeley, May 2010.