Parallel LDA Through Optimized Synchronous Communication Methods

Bingjing Zhang, Bo Peng, Judy Qiu
Computer Science Department
Indiana University
Bloomington, IN, USA
zhangbj, pengb, xqiu@indiana.edu

***Abstract*—Selecting an appropriate distributed processing framework can be difficult for developers building large-scale machine learning applications. That is because all these tools provide various kinds of parallelism patterns and suggest different communication strategies to synchronize local and global model data distributed among parallel nodes. There is no clear answer to determine which strategy might be suitable depending on the data and application. Taking Latent Dirichlet Allocation (LDA) as an example, contemporary implementations often choose asynchronous communication methods to synchronize the model data. However, our observations show that the asynchronous communication still has very high overhead and the characteristics of the LDA training datasets encourage us to use optimized synchronous collective communication methods instead. The results show that with data parallelism only, our "lda-lgs" implementation can be (%) faster compared to Yahoo! LDA. With model parallelism, our "lda-rtt" implementation has similar speed compared with Petuum LDA on a uni-gram model with 1 million words and 10k topics but (%) faster on a bi-gram model with 20 million words and 500 topics.**

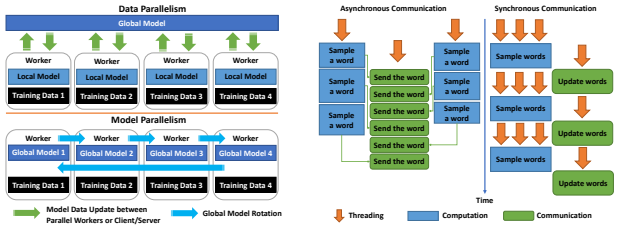
I. Introduction

One challenge of parallel machine learning applications is that while training data can be split into parallel workers, the model data that all local computations depend on is growing progressively and generates significant synchronization overhead. Currently two types of parallelism are used to solve this problem (see Fig. 1a):

Data Parallelism The global model is distributed on a set of servers or on existing parallel workers. Each worker samples on a local model and updates it through the synchronization between local models and the global model.

Model Parallelism In addition to using data parallelism, the global model data is split between parallel workers and rotated during the sampling.

In LDA [1], the model synchronization is important because a faster communication method not only reduces the resulting overhead, but also speeds up the model convergence rate, shrinks the model size, and shortens the computation time in later iterations. Though both synchronous and asynchronous methods (see Fig. 1b) can cause the model to converge without affecting the correctness of the algorithm, it is unclear which strategy performs better for LDA applications. Asynchronous communication is popular because it avoids the overhead of global waiting between parallel workers and that of local waiting between computation threads and communication threads.



(a) (b)

Fig. 1. (a) Data Parallelism vs. Model Parallelism and (b) Asynchronous Communication vs. Synchronous Communication in LDA

In data parallelism, asynchronous communication allows local computation to continue without waiting for the completion of updating the global model from all parallel workers per iteration. In model parallelism, though model rotation is synchronous, per word sampling and sending can still overlap without waiting on each worker, demonstrating asynchronous communication.

However, after studying the characteristics of LDA data, we have identified that the counts of each word in the training documents fall under the power-law distribution. As a result, when data parallelism is used, many words in the global model will display on all the workers' local models, and this generates "one-to-all" communication patterns during the synchronization. Similarly, in model parallelism, as the size of the global model data expands, each worker needs to handle more data transference. These observations inspired us to apply routing optimized synchronous communication operations to improve the the LDA model update speed.

Our synchronous communication methods utilize the model data distribution characteristics and routing optimization in conjunction. Furthermore, we overlapped the computation and communication steps to reduce the overhead of the global/local waiting. These ideas are implemented in Harp [2], a collective communication library on Hadoop. Harp has already integrated several collective communication patterns from different parallel processing frameworks in a unified abstraction. However, all the current patterns cannot abstract either the local/global model synchronization in data parallelism or the model rotation in model parallelism. As such, we abstracted three other communication patterns called "syncLocalWithGlobal",

"syncGlobalWithLocal", and "rotateGlobal" in which our new ideas are embedded. The new patterns are very generalizable so that they can be applied not only to LDA applications but also to many other machine learning applications. We implemented one LDA application which uses "syncLocalWithGlobal" and "syncGlobalWithLocal" to perform data parallelism and another which uses "rotateGlobal" to perform model parallelism. We compared our implementations with other implementations based on asynchronous communication methods, such as Yahoo! LDA [3] and Petuum LDA [4], on several datasets. The results show that optimized synchronous communication methods can reduce communication overhead and improve model convergence speed.

The following sections describe: the cost model of LDA algorithm (Section 2), the synchronous communication methods (Section 3), the implementation of Harp-LDA (Section 4), the performance results of our implementation (Section 5), the related work on parallel LDA (Section 6), and our conclusions (Section 7).

II. Cost Model

A. LDA model

Latent Dirichlet Allocation (LDA) is a generative probabilistic data modeling technique. Training data are abstracted as a document collection, where each document is a bag of words. LDA models the data by introducing latent topics, which try to capture the underlining semantic connections and structures inside the data. In LDA model, a document is a mixture of latent topics and each topic is a multinomial distribution over words. In the generative process, for document j, we first draw a topic distribution θ_j from a Dirichlet with parameter α. Then for each word i in this document, we draw a topic $z_{ij} = k$ from the multinomial distribution with parameter θ_j. Finally, word x_{ij} is drawn from a multinomial $\phi_{wk|k=z_{ij}}$, which also derives from a Dirichlet with parameter β. Here, the words x_{ij} are observed variables, θ, ϕ, z are latent variables, and α and β are hyper parameters.

The purpose of LDA inference is to compute the posterior distribution of the latent variables given the observed variables. There are many approximate inference algorithms. In a practice on large data, Collapsed Gibbs Sampling (CGS) [5] displays high scalability. Collapse is a procedure used to integrate out θ,ϕ and sample only the latent variables z. Gibbs Sampling is one kind of Markov chain Monte Carlo algorithm for inference. There are three phases: initialize, burn-in and stationary.

In initialize, each word is initialized by a random topic denoted as z_{ij}. Afterwards it begins to reassign topics to each word w_{ij} according to the conditional probability of z_{ij}, which then calls sampling.

$$p\left(z_{ij} = k \mid z^{\neg ij}, x, \alpha, \beta\right) \propto \frac{N_{wk}^{\neg ij} + \beta}{\sum_w N_{wk}^{\neg ij} + V\beta} \left(N_{kj}^{\neg ij} + \alpha\right) \tag{1}$$

Here, superscript $\neg ij$ means that the corresponding word is excluded in the counts. V is vocabulary size. N_{wk} is the count of word w assigned to topic k, and N_{kj} is the count of topic k assigned in document j, which are sufficient statistics for the latent variable θ and ϕ. The latent variables can be represented by three matrices Z_{ij}, N_{wk} and N_{kj}, which are model data. Intuitively, by equation(1), with higher probability a word will be assigned to the topic that has been assigned to it's co-occurring words. So, sampling by the latest model data of co-occurring words is critical for convergence, that is why synchronization is so important in parallel LDA trainer.

Hyper parameters α and β are also called concentration parameters, which control the topic density in the final model. The larger the α and β, the more topics can be drawn into a document and assigned to a word, and the more non-zero cells in each row of the N_{wk} and N_{kj} matrices. Although a useful LDA trainer often has the feature of α and β optimization dynamically tuned to fit the training data, in this paper, we skip such a feature and use symmetric α and β both fixed to a common used value 0.01 to exclude the complex effects on performance caused by their dynamics.

Latent variables will gradually converge in the process of iterative sampling. This is the phase where burn-in occurs and finally reaches the stationary state. From that point, we can draw samples from the sampling process and use them to calculate the posterior distribution.

To evaluate the quality of the final model learned by LDA, held-out testsets are often used, taking likelihood or perplexity as the accuracy metrics. In this paper, we just use the model data likelihood on the training dataset to monitor the convergence of LDA trainer, which is consistent with the held-out test set results in our experiments, only much faster.

Sampling on z_{ij} in CGS is a strictly sequential process. AD-LDA[6] is the seminal work allowing us to relax this sequential sampling requirement. It assumes that the dependence between one topic assignment z_{ij} and another z_{ij} is weak in the case that different words in different documents are sampled concurrently. In AD-LDA, training data are partitioned into n subsets, with n Gibbs Samplers running parallel on each collection, and each sampler synchronizes its model data with others at certain time points. This parallel version still produces a useful model. This established the foundation of large-scale parallel implementations of CGS of LDA trainers on large-scale data in practice.

B. Performance Factors

Many factors are related to the performance of a LDA trainer.

Sampling Algorithm Computation complexity of a sampling algorithm basically determines the overall performance. Although there is a $\mathcal{O}(1)$ sampling algorithm, LightLDA [7], proposed in the literature, we still focus on SparseLDA [8], which is an optimized CGS sampling algorithm mostly used in the state-of-the-art LDA trainers, in order to make a broader

comparison. SparseLDA splits the equation (1) into three parts:

$$p\left(z_{ij}=k \mid rest\right) \propto \frac{N_{wk}^{\neg ij}(N_{kj}^{\neg ij}+\alpha)+\beta * N_{kj}^{\neg ij}+\alpha\beta}{\sum_w N_{wk}^{\neg ij}+V\beta} \tag{2}$$

The denominator is a constant when sampling on one word. The third part of the numerator is also a constant; the second part is non-zero only when N_{kj} is non-zero, and the first part is non-zero only when N_{wk} is non-zero. In naive CGS sampling, the conditional probability will compute K times, while in SparseLDA, the computation can be decreased to non-zero items number in N_{wk} and N_{kj}, which are much smaller than K on average.

We found that in practice, the sampling performance is more memory bounded than computation bounded, for the computation is very simple and memory access to two large matrices is not by its nature cache friendly. Furthermore, CGS has a feature of exchangeability that permits the order of word sampling to be changed. In practice, sampling can take the order by row or column on the document-word matrix. Equation(2) is the form optimized for row order, called sample-by-doc. In this case, N_{kj} can be cached for the words in the same row, and the computation complexity in terms of amotized random memory access time is $\mathcal{O}(\sum_k \mathbb{1}(N_{wk} \neq 0))$. Symmetrically, sample-by-word will have the complexity of $\mathcal{O}(\sum_k \mathbb{1}(N_{kj} \neq 0))$.

Parallelism Strategy Data partition on the training data, which is a document-word matrix, can be done either in the rows or the columns. If data are partitioned by rows, each subset data has its local z, N_{kj}, N_{wk} model data and only N_{wk} needs to be synchronized with others. In general applications, the row number is much larger than the column number, so partition by rows will get smaller model data size. Here we only call the shared word-topic matrix as model data.

There are many possible communication strategies which control how to do model data synchronization among parallel units. Modern cluster has two levels of parallel units; one is distributed processes on an inter-nodes level and the other is multi-threading inside the node level. In this paper, we focus on the inter-nodes level by exploring the differences among the communication strategies.

Cluster configurations include nodes number N and networking bandwidth B, memory size M for each node, and thread number T for each node. As manycore technology brings more powerful machines to bear for complicated computation applications, large-scale machine learning applications will definitely benefit as a result. Relatively small numbers of N with a large number of T can reach high scale parallelism, which is more like a traditional HPC cluster than a cloud cluster.

Data Property Training data can be characterized by the total numbers of tokens, denoted as W, and number of documents, denoted as D. The model data N_{wk} is a $V * K$ matrix and N_{kj} is a $D * K$ matrix, where V is the vocabulary size and K is the topic number.

LDA is an iterative algorithm. It keeps sampling on the training data and updating (synchronizing) the model data until it converges. In the computation part, one iteration is one pass on sampling the training data. In the synchronization part, one iteration is one pass to synchronize all the model data. As we described above, both parts are highly related to the model data size, not in terms of the matrix dimension but the non-zero items count.

C. Model Data

Model Size Power law distribution is a general phenomenon. It has another equal form for text data as Zipf's law, where the frequency of a word is proportional to the reciprocal of its rank.

$$freq(i) = C * i^{-\lambda} \tag{3}$$

here, i is word rank, and λ is near 1.

There are a total of V unique words in the training data. We then have:

$$\begin{aligned} W &= \sum_{i=1}^{V}(freq(i))) = \sum_{i=1}^{V}(C * i^{-\lambda}) \\ &\approx C * (ln(V) + \gamma + \frac{1}{2V}) \end{aligned} \tag{4}$$

If λ is 1, this is the partial sum of harmonic series which have logarithmic growth, where γ is the EulerMascheroni constant ≈ 0.57721.

Model data, $V * K$, is a very large but sparse matrix. In a general setting, V is 1M, K is 1K, while for big models it can even reach 1M*1M. But the non-zero cell count of the matrix is the true model size, denoted as S, $S << V * K$.

In the initialization phase of CGS, word-topic count matrix is initialized by random topic assignment for each work. So the word i will get $max(K, freq(i))$ non-zero cells. if $freq(J) = K$, $J = C/K$, we get:

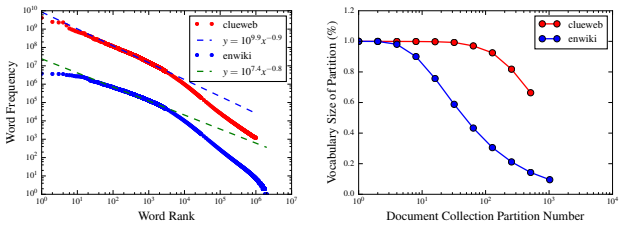
$$\begin{aligned} S_{init} &= \sum_{i=1}^{J} K + \sum_{i=J+1}^{V} freq(i) = W - \sum_{i=1}^{J} freq(i) + \sum_{i=1}^{J} K \\ &= C * (lnV + lnK - lnC + 1) \end{aligned} \tag{5}$$

The true model size S_{init} is logarithmic to matrix size $V * K$. This does not mean S_{init} is small, for the constant $C = freq(1)$ can be very large; even $C * ln(V * K)$ can be huge. But it basically means that an increase of dimension in the model will not increase the model data size dramatically.

With the progress of iterations and algorithm convergence, the model data size will shrink. The concentration parameters α and β control the final sparsity of the topic distribution. When a stationary state is reached, the average count value will drop to a certain small constant ratio of K, with the constant δ determined by the properties of the training data itself.

$$S_{final} = mean(word - topiccount) * V = \delta * K * V \tag{6}$$

Model Data Partition After training data is partitioned to each node of the cluster, a local model data S' will be built up and used in local computation. This local model data should



(a) (b)

Fig. 2. Model Size of (a) Zipf's Law and (b) Vocabulary and Data Partition

synchronize with global model data S frequently to make the training process converge. In fact, the synchronization frequency is highly relevant to the final model accuracy.

This data partition strategy can decrease local training data W' linear to node number N. Therefore, we get $W' = W/N$. For computations proportional to the total word number W', this strategy is friendly to computation, and the more nodes we have, the better performance we can expect. And, assuming $C' = C/N$, the actual local model size S'_{init} is:

$$\begin{aligned} S'_{init} &= C' * (lnV' + lnK - lnC' + 1) \\ &\leq \frac{C}{N}(lnV + lnK - lnC + 1 + lnN) \\ &\leq \frac{S}{N} + \frac{C}{N}lnN \end{aligned} \tag{7}$$

In general configurations lnN is smaller than $lnV + lnK - lnC + 1$, so local model size S'_{init} is no more than $\frac{2}{N}S_{init}$. The initialized local model data size is controllable by data partition.

But when model data synchronization begins, all words in the local vocabulary need to fetch the corresponding global model data. The local vocabulary size V' will then determine both the communication data volume and local model size in the burn-in phase, which becomes the problem.

It is clear that when documents are partitioned to N nodes, every word with frequency larger than N will get a high probability occurring on each node. If at rank L, $freq(L) = N$, we get: $L = \frac{W}{(lnV+\gamma)*N}$ On the "enwiki" dataset, W=1B, V=1M, N=100, we get $L = 0.69V$; on the "clueweb" dataset, W=10B, V=1M, N=100, $L > V$. For a reasonably large training dataset, L should be easily larger than V, which means it will send/receive and hold almost all the global model data locally.

So, we conclude that because the power law distribution of data exists, general data parallelism can help to distribute training data among nodes and parallelize the computation tasks accordingly, but it cannot effectively control the volume of the model data movements among nodes. When dealing with larger data and larger models, simply deploying more nodes will not prove an effective solution, for model data synchronization will eventually become a bottleneck.

D. Experiments

We first validate Zipf's law of word distribution on "clueweb" and "enwiki" datasets, where the top 1M most frequent words are selected(see Fig. 2a). They both show considerable matching results, especially in the word region with high frequency. In the preprocess step for LDA trainer, stop words and low frequency words are often removed. This makes the slope a bit more flat, and the actual model denser than expected from equation(5). In Fig. 2b, we represent the difficulty of controlling the vocabulary size by random partition of document collection. When 10 times more partitions are introduced, there is only a sub-linear portion decrease of the vocabulary size in each partition compared to the total one; e.g. on the "clueweb" dataset, each partition gets 92.5% vocabulary size when data is randomly distributed to 128 nodes. The "enwiki" dataset is about 12 times smaller than "clueweb", and it gets 90% at 8 nodes, keeping the similar ratio. This figure shows the local models will not be the same size of the global one, but also not much smaller.

III. Synchronous Communication Methods

Past research has shown that collective communication operations are indispensable in iteration-based machine learning algorithms. Chu et al. [9] mentions that many machine learning algorithms can be implemented in MapReduce systems [10]. The underlying principle of this conclusion is that each iteration in the algorithm is dependent on the synchronization of the local models computed on each worker at the last iteration. However, MapReduce systems only provide a fixed "shuffling" communication pattern. Thus, in Harp, a separate collective communication abstraction layer provides a set of data abstractions and related collective communication operation abstractions.

For LDA, both data parallelism and model parallelism benefit from optimized synchronous communication methods. In data parallelism, "one-to-all" communication patterns play a crucial role in the synchronization to enable the optimization of the communication performance with collective communication operations. In model parallelism, using collective communication can maximize bandwidth usage between a worker and its neighbors in shifting the model partitions.

A. The Abstraction Of Global/Local Data Synchronization

Considering the sparsity of the local model data distribution on workers, the collective communication optimization, and the existing collective communication abstractions in Harp, we added two other data abstractions and related new collective communication operations.

The two types of data abstractions are the global table and the local table. The concept "table" has been defined in previous Harp collective communication abstractions [2]. Each table may contain one or more partitions, and the tables defined on different workers are associated in order to manage a distributed dataset. In global tables, each partition has a unique ID and represents a part of the whole distributed dataset; but in local tables, partitions on different workers can share the same partition ID. Each of these partitions sharing the same ID is considered a local version of a partition in the full distributed dataset.

We defined three communication operations on global tables and local tables, with the first two being paired operations. First, "syncGlobalWithLocal" uses the data in local tables to synchronize the data in global tables. This operation will reduce the partitions from local tables to the global table. Secondly, "syncLocalWithGlobal" uses the data in global tables to synchronize local tables. Based on the needs of partitions in local tables, this operation will redistribute the partitions in the global table to local tables. If one partition is required by all the workers, it will be broadcasted.

Lastly, "rotateGlobal" will consider workers in a ring topology and shift the partitions in the global table owned by one worker to the right neighbor worker and then receive the partitions from the left neighbor. When the operation is completed, the contents of the distributed dataset in the global tables won't change, but each worker will hold a different set of partitions. Since each worker only talks to its neighbors, "rotateGlobal" can transmit global data in parallel without any network conflicts.

B. The Applicability of Synchronous Communication Methods

"syncGlobalWithLocal" and "syncLocalWithGlobal" are abstracted from data parallelism, and "rotateGlobal" is abstracted from model parallelism. However, these operations are not limited to the communication patterns in parallel LDA. Instead, they can be applied to many other machine learning algorithms with big model data.

A matrix can be drawn to describe each worker's requirements on the global model data in the parallel computation per iteration. In this matrix, each row represents a worker, each column represents a global data partition, and each element shows the requirements of the partition in the local computation. Based on the density of this computation relation matrix, we can choose proper operations in different applications. If the matrix is dense, we suggest using the "rotateGlobal" operation. Using k-means clustering as an example, the global model data are the centroids, and the local computation needs all the centroids data. Thus "rotateGlobal" allows each worker to access all the centroids data efficiently. If the matrix is sparse, using "syncGlobalWithLocal" and "syncLocalWithGlobal" is a superior solution. For example, in graph algorithms such as PageRank, the global model data are the vertices' page-rank values and counts of out-edges. The local computation goes through each edge and calculates the partial result of the new page-rank values. Then "syncGlobalWithLocal" can be used to update global page-rank values. In the next iteration, we can use "syncLocalWithGlobal" to fetch the new global page-rank values to each local computation.

IV. Harp-LDA Implementation

A. Partition Training Data And Initialize Model Data

For the training data, we split the documents into files evenly. For the model data, since words with high frequency can dominate the computation and communication, we partition the global model based on the frequency of words in the training dataset. During the preprocessing of the training data, each word is given an ID based on their frequency starting from 0. The lower the occurrence of the word, the higher the ID. Then we partition the words' topic counts using range-based partitioning. Assuming each partition contains m words, Partition 0 contains words with IDs from 0 to $m-1$, and Partition 1 contains words with IDs from m to $2m-1$, and so on. As a result, the partitions with low IDs contain the words with the highest frequency. The initial global model is generated by randomly assigning each token to a topic and aggregated through "syncGlobalWithLocal". The mapping between partition IDs and worker IDs is calculated based on the modulo operation. Assuming there is a worker with ID w among a total of N workers, the partitions contained on this worker are Partition w, Partition $w+N$, Partition $w+2N$, and so on. In this way, each worker contains a number of words whose frequencies rank from high to low.

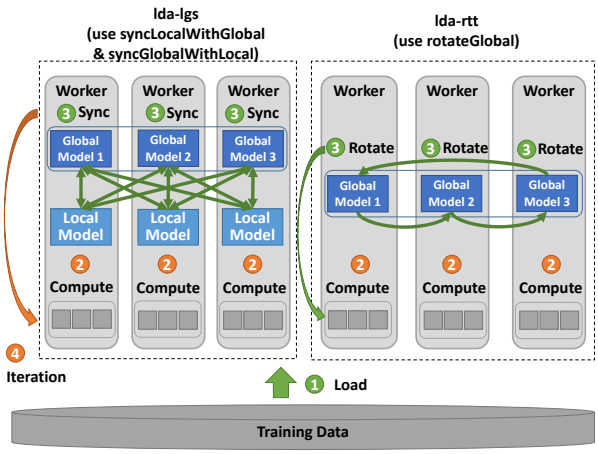


Fig. 3. Internode Parallelism (data loading ⟨step 1⟩ and iteration ⟨step 4⟩ are common procedures for both implementations)

B. Internode Parallelism

During iterations of the sampling, we use two different approaches to update the global model which results in two implementations (See Fig. 3). One implementation, named "lda-lgs", follows data parallelism and uses "syncGlobalWithLocal" paired with "syncLocalWithGlobal" operations. The other implementation, named "lda-rtt", follows model parallelism and uses "rotateGlobal" operation.

During the sampling of "lda-lgs", each worker updates the local model and tracks the difference generated in another table. Once the sampling is done, "syncGlobalWithLocal" operation is used to update the global model with the changes of the local model. "syncLocalWithGlobal" operation is then used to download new local model data from the updated global table. At the end of the iteration, the sum of word counts for each topic is calculated with "allreduce" operation [11].

In "lda-rtt", each worker will first conduct sampling with the global model partitions owned by itself and update them

directly. Then it will call "rotateGlobal" operation to send the updated model data to the right neighbor and receive model partitions from the left neighbor. Once all partitions of the global model are received and processed, the sampling of one iteration is completed. Similar to "lda-lgs", "allreduce" operation is used at the end of the iteration to update the global sum of word counts on all topics.

C. Overlap Communication with Computation

Synchronous communication methods are often criticized for generating much overhead and making all workers wait for the completion of synchronization. We approached this problem in three steps. The first step is to balance the communication load on each worker through partitioning the global model based on word frequencies. The second step is to improve the speed with optimized collective communication. Here we discuss the third step, which is overlapping computation and communication in execution.

In "lda-rtt", we slice the global model partitions held on each worker into two sets. Slicing is conducted by first sorting the partition IDs in ascending order and then assigning the partitions to the two slices in alternate order. As a result, each slice will contain words with both high and low frequencies. We can view the whole process as splitting one global data table distributed across workers into two global data tables. During the sampling, when a worker finishes processing the first slice, it uses another thread to rotate this slice. At the same time, it continues processing the second slice. Once the second slice is processed, the first slice has been rotated and is ready for further processing. When both slices have finished one round of rotation, the sampling of one iteration is over. The overlapping between computation and communication occurs when the worker processes one slice and rotates another slice at the same time.

In "lda-lgs", we split the local data table into two slices. During the sampling, when each worker samples one slice, it asks another thread to synchronize the other slice through the paired "syncLocalWithGlobal" and "syncGlobalWithLocal" operations. We map partitions based on their IDs into slices so that local partitions with the same ID are guaranteed to be synchronized in iterations.

D. Innernode Parallelism

In Harp-LDA, we use the "Computation" component provided by Harp to manage the multi-threading sampling within one worker. The sampling process follows a SparseLDA algorithm and can be performed in varying order. One approach is to go through each document and sample every token. Another is to go through each word and sample its occurrences in each document. To keep the sampling order consistent between implementations and remain fair in later performance comparisons in the experiments, we sample by documents in "lda-lgs" as Yahoo! LDA and sample by words in "lda-rtt" as Petuum LDA. Notice that when sampling by words, we also balance the computation load per thread by assigning words based on their frequencies.

The local model is shared between threads. When sampling by documents, the word-topic model is required to access with locks. Symmetrically, when sampling by words, the document-topic model is required to access with locks. We provide a read lock and a write lock on each document/word's topic count map. Before sampling a token, all its document/word topic counts are read out, and after sampling the updates are written back. If the next token for sampling is the same word, the sampling thread will keep using the thread local cached topic counts to avoid repeating fetching the shared data. During the update, we separate "updating an existing topic entry" and "adding a count to a new topic entry". In "updating an existing topic entry", because the map structure is not altered during updating and reading a primitive integer is an atomic operation in modern x86 architecture, it is safe to execute "read" and "update" concurrently with a shared read lock. But in order to ensure the correctness of the topic count values, "update" operations are still required to be exclusive from each other. In the operation of "adding a count to a new topic entry", since the map structure is modified, we have to use a write lock.

Though the concurrency is greatly improved, our current implementation is still slower compared with Yahoo! LDA and Petuum in the first iteration of sampling. This could be caused by the difference on the implementation language (Java/C++) and the performance of the data structure (primitive int based hashmap [12]/primitive int array). As many-core architecture is becoming more common, high performance concurrent sampling with many-threads is a challenge to all the implementations. However, in this paper our aim is not to provide the fastest LDA implementation but to show the advantages of using synchronous communication methods in converging LDA model compared with asynchronous communication methods.

V. Experiments

A. Experiment Settings

The tests are done on Juliet cluster [13]. Juliet cluster contains 32 18-core 72-thread nodes and 96 24-core 48-thread nodes. All the nodes have 128GB memory and are connected with two types of networks: 1Gbps Ethernet (eth) and 16 Gbps Infiniband (ib). For testing, we use 31 18-core nodes and 69 24-core nodes to form a cluster of 100 nodes with 40 threads on each for computation. Most tests are done with Infiniband through IPoIB support unless otherwise specified.

Several datasets are used (see Table I). The total number of model parameters is kept as 10 billion on all the datasets. α and β are both fixed at 0.01.

We test several implementations on these datasets (see Table II). We compare synchronous communication methods with asynchronous communication methods on both the model parallelism and the data parallelism. By studying the convergence speed and the execution time, we learn how the difference in communication methods affects the performance of LDA.

TABLE I
TRAINING DATA SETTINGS USED IN THE EXPERIMENTS

Dataset	enwiki	clueweb	bi-gram	gutenberg
Num. of Docs	3.8M	50.5M	3.9M	26.2K
Num. of Tokens	1.1B	12.4B	1.7B	836.8M
Vocabulary	1M	1M	20M	1M
Doc Len. AVG/STD	293/523	224/352	434/776	31879/42147
Lowest Word Freq.	7	285	6	2
Num. of Topics	10K	10K	500	10K
Init. Model Size	2.0GB	14.7GB	5.9GB	1.7GB

Note: Both "enwiki" and "bi-gram" are English articles from Wikipedia. "clueweb" is 10% English web pages from ClueWeb09 [14]. And "gutenberg"are English books from Project GutenBurg [15].

TABLE II
LDA IMPLEMENTATIONS USED IN THE EXPERIMENTS

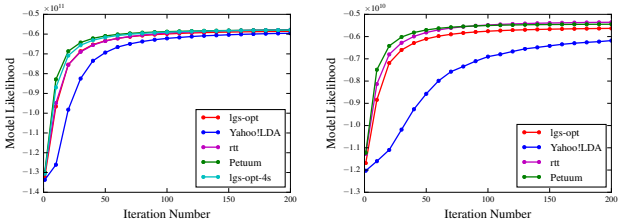
Data Parallelism	
lgs	This refers to "lda-lgs" implementation. This version uses SparseLDA algorithm and sample by documents but no routing optimization.
lgs-opt	This version is similar to "lgs" and uses routing optimization.
lgs-opt-4s	This version is similar to "lgs-opt" but the training data on each worker is divided into 4 slices. During each iteration, when sampling a slice, a full model synchronization is performed.
Yahoo! LDA	This refers to the master branch on GitHub [3]. It uses SparseLDA algorithm and samples by documents.
Model Parallelism	
rtt	This refers to "lda-rtt" implementation. This version uses SparseLDA algorithm and samples by words.
Petuum	This refers to version 1.1 [4]. It uses SparseLDA algorithm and samples by words.

B. Convergence Speed Per Iteration

Firstly we compare the convergence speed of the LDA word-topic model on iterations by analyzing model results learned on Iteration 1, 10, 20, 30... 200. Because all the training data are sampled once in one iteration, it is fair to measure the performance of the model convergence without considering the performance difference in computation.

On the "clueweb" dataset (see Fig. 4a), Petuum has the highest model likelihood on all the iterations. Though "rtt" also uses the model parallelism, due to its preference of using the thread-local data and not the up-to-date local shared model, the convergence speed is slower. "rtt" and "lgs-opt" have similar convergence speeds and their lines in the chart are close to overlapping. This is different from "lgs-opt", where the convergence speed of "lgs-opt-4s" is as high as Petuum. This shows that increasing the times of model synchronization helps the convergence speed. Yahoo! LDA has the slowest convergence speed because its asynchronous communication does not guarantee a full model synchronization in one iteration.

On the "enwiki" dataset (see Fig. 4b), as before, Petuum has the highest accuracy out of all iterations. "rtt" converges to the same model likelihood level as Petuum at Iteration 200. "lgs-opt" has slower convergence speed but still achieved high model likelihood, while with Yahoo! LDA both have the slowest convergence speed and achieve the lowest model likelihood at Iteration 200.



(a) (b)

Fig. 4. Model Convergence of (a) "clueweb" And (b) "enwiki" On Iterations

All these results show that when model update rate is increased (either using the model parallelism or using multiple-time model synchronization in the data parallelism), the model converges faster.

C. Performance Analysis on Data Parallelism

We compare the model convergence speed on "lgs" and Yahoo! LDA by injecting the real execution time on iterations. On the "clueweb" dataset, we first show the convergence speed based on the elapsed execution time (see Fig. 5a). Yahoo! LDA need more time to obtain the model result of Iteration 1 due to its slown model initialization. Since model initialization is mainly communication rather than computation and it cannot be overlapped with sampling, this shows Yahoo! LDA has huge overhead on the communication end. In later iterations, though "lgs" converges faster, Yahoo! LDA catches up after 30 iterations. The reason is that our computation is slow and we only allow one model synchronization per iteration, while Yahoo! LDA does not have this restriction and allow multiple instances of synchronization whenever possible. Because the computation is quite long and the network is idle most of the time, we can increase the time of model synchronization per iteration by dividing the training documents on each worker into multiple slices and synchronizing the model data when sampling each slice. "lgs-opt-4s" shows that although the execution time on 200 iterations is still slightly longer than Yahoo! LDA, it can obtain higher model likelihood and keep faster convergence speed in the whole execution.

Due to the slowness in the local computation, our implementations show much higher iteration execution time at the first iteration compared with Yahoo! LDA (see Fig. 5b). But with optimized synchronous communication methods, we quickly reduce the difference and could even run faster than Yahoo! LDA on some iterations. Similar results are also shown on the "enwiki" dataset. "lgs-opt"not only achieves higher model likelihood but also has faster model convergence speed in the while execution (see Fig. 5d). Though our execution time at Iteration 1 is twice as slow as Yahoo! LDA, later it takes less execution time per iteration than Yahoo! LDA (see Fig. 5e). It is caught up with Yahoo! LDA only when the models in both applications converge to a similar likelihood level.

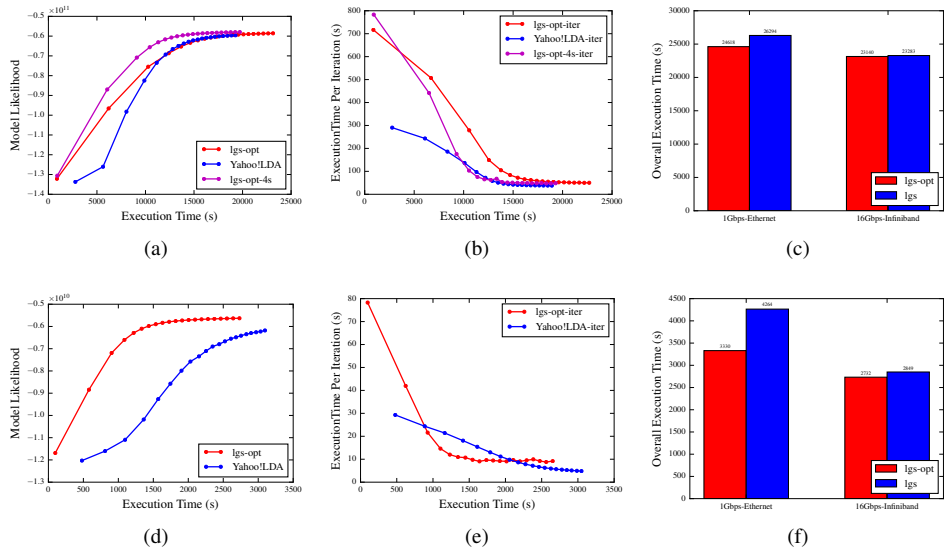


Fig. 5. Performance comparison on Data Parallelism between "lgs" and Yahoo! LDA (a) Elapsed Execution Time vs. Model Likelihood on "clueweb" (b) Elapsed Execution Time vs. Iteration Execution Time on "clueweb" (c) Total 200-Iteration Execution Time with Routing Optimization vs. One without Routing Optimization on "clueweb" with ib/eth (d) Elapsed Execution Time vs. Model Likelihood on "enwiki" (e) Elapsed Execution Time vs. Iteration Execution Time on "enwiki" (f) Total 200-Iteration Execution Time with Routing Optimization vs. One without Routing Optimization on "enwiki" with ib/eth

In addition, we examine the effectiveness of using routing optimization in our "lgs" solution. Fig. 5c shows the results of 200 iteration executions on "clueweb" and Fig. 5f shows the results on "enwiki". On Ethernet, "lgs-opt" is obviously faster than "lgs". But with Infiniband, due to its high bandwidth, the performance is very close to one another.

D. Performance Analysis on Model Parallelism

Here we compare "rtt" and Petuum on 3 different datasets: "clueweb", "bi-gram" and "gutenburg". Because both these implementations use model parallelism, the performance difference is caused by the execution speed per iteration.

On the "clueweb" dataset, both implementations achieve similar model likelihood with similar execution times after 200 iterations (see Fig. 6a). But the first 10 iterations still show that "rtt" has high computation time compared with Petuum (see Fig. 6b). However its overhead on communication per iteration becomes lower than Petuum. When the execution arrives at the final 10 iterations, while the computation overhead per iteration in "rtt" is still higher, the whole execution time per iteration becomes lower (see Fig. 6c). The trend of this change is shown in Fig 6d.

Unlike our "rotateGlobal" operation which batches transmission of model data partitions, Petuum sends model data word by word asynchronously. This could cause high communication overhead. On a "bi-gram" dataset, the results show that when the number of words in the model gets high, Petuum cannot perform well. Due to the high overhead in communication, the convergence speed is very slow and Petuum cannot even continue executing after 60 iterations due to an out of memory error (see Fig. 6e). Due to the communication overhead, Fig. 6f and Fig. 6g show that in the first/final 10 iterations, Petuum always has higher execution time per iteration compared with "rtt". The trend of this phenomenon is shown in Fig. 6h.

Though the data size of "gutenburg" is similar to "enwiki", it is clear that there is a difference in execution speed per iteration (see Fig. 6i). High standard deviation indicates that the iteration execution time per worker varies a lot. Not like the results on "bi-gram" where Petuum's performance suffers from the communication overhead, here it suffers from waiting for the slowest worker. The reason is that "gutenburg" contains many long documents and results in unbalanced training data distribution on the workers. Besides, when sampling by words, frequent access to the shared huge doc-topic model leads to inefficient concurrent sampling. However, "rtt" is not much affected because it prefers using thread-local data in concurrent sampling and balances per-thread computation through assigning words to threads based on the frequencies. Fig. 6j, Fig. 6k, and Fig. 6l display that the unbalanced computation in Petuum results in high overhead per iteration. Since in model parallelism, model rotation is a synchronous operation, this experiment demonstrates that unbalanced computation on workers causes huge overhead in global waiting and results in high iteration execution time. As a result, when applying synchronous communication methods, computation load balancing should be carefully considered.

VI. Related Work

Prior research has studied the parallelization of the LDA algorithm extensively. Some studies focused on using the Collapsed Variational Bayes (CVB) algorithm [1]. Mahout LDA [16] and Spark LDA [17] both use this algorithm. However, research also shows that this approach leads to high memory consumption and slow convergence speed [6][18].

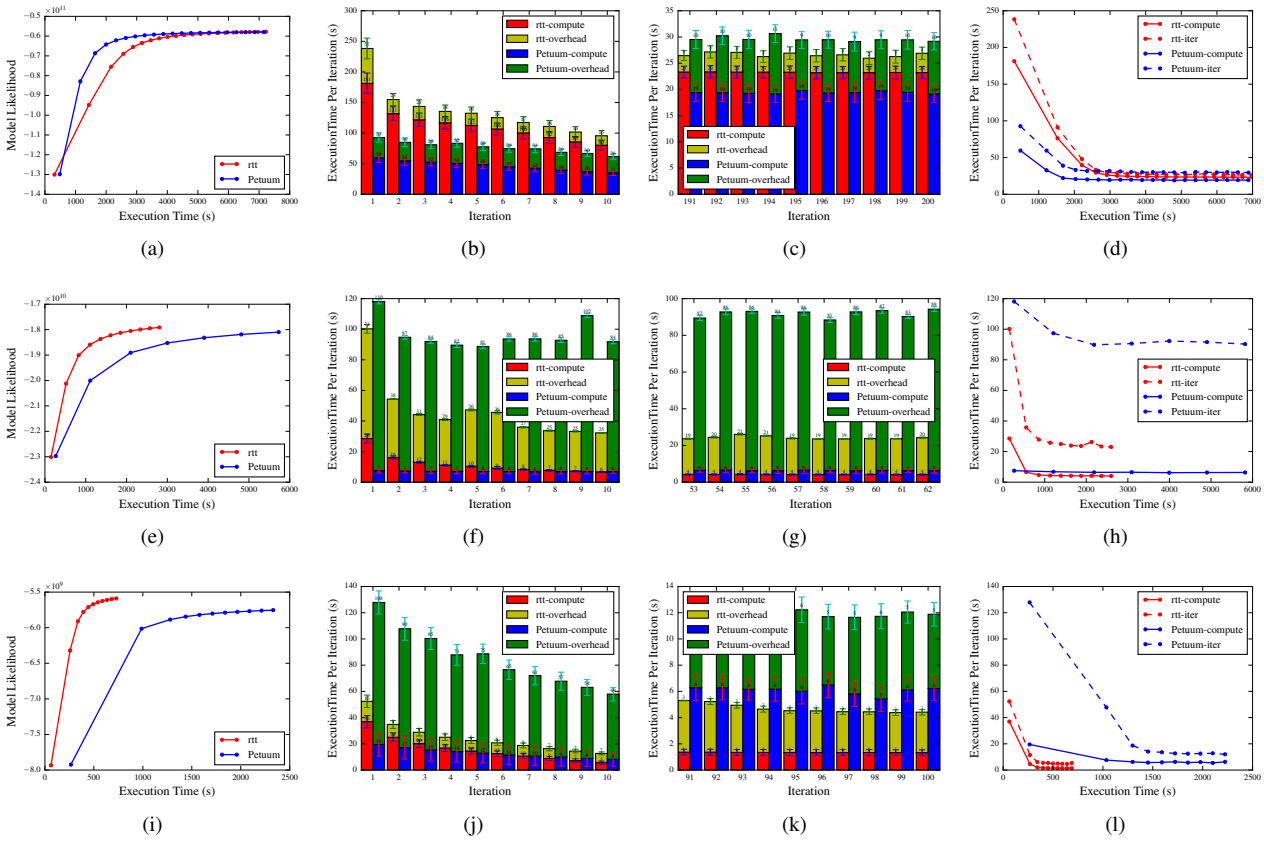


Fig. 6. Performance comparison on Model Parallelism between "rtt" and Petuum (a) Elapsed Execution Time vs. Model Likelihood on "clueweb" (b) First 10 Iteration Execution Times on "clueweb" (c) Final 10 Iteration Execution Times on "clueweb" (d) Elapsed Execution Time vs. Iteration Execution Time on "clueweb" (e) Elapsed Execution Time vs. Model Likelihood on "bi-gram" (f) First 10 Iteration Execution Times on "bi-gram" (g) Final 10 Iteration Execution Times on "bi-gram" (h) Elapsed Execution Time vs. Iteration Execution Time on "bi-gram" (i) Elapsed Execution Time vs. Model Likelihood on "gutenburg" (j) First 10 Iteration Execution Times on "gutenburg" (k) Final 10 Iteration Execution Times on "gutenburg" (l) Elapsed Execution Time vs. Iteration Execution Time on "gutenburg"

Other studies use the CGS algorithm (see Table III). PLDA [19] is such an implementation. There are two versions of PLDA, one based on MPI [20] using the "allreduce" operation [11], and the other based on on MapReduce[10][21] using "shuffle" operation.

Yahoo! LDA [22][23] uses the CGS algorithm with SparseLDA optimization, and its architecture is client-server with asynchronous communication. Local models are distributed in the star model, and local computation threads use optimized locking mechanisms when accessing the shared local model. The synchronization between local models and the global model is done through asynchronous delta aggregation.

Dato [24] uses the GAS model [25] to implement the LDA algorithm [26]. Currently, it uses a CGS algorithm without SparseLDA optimization. GAS model's edge-based computation patterns cause the training data to be partitioned based on document-word pairs instead of the documents. As a result, during the sampling process, both the topic counts of words and documents have to be gathered and updated. This results in additional communication costs in synchronization.

Peacock [18] uses a hierarchical distributed architecture to organize the LDA computation. The first layer uses the

TABLE III
LDA WORK USING CGS ALGORITHM

App. Name	Algorithm	Parallelism	Comm.
PLDA	CGS (sample by docs)	D. P.	allreduce (sync)
Dato	CGS (sample by doc-word edge)	D. P.	GAS (sync)
Yahoo! LDA	CGS (SparseLDA & sample by docs)	D. P.	client-server (async)
Peacock	CGS (SparseLDA & sample by words)	D. P. (M. P. in local)	client-server (async)
Parameter Server	CGS (combined with other methods)	D. P.	client-server (async)
Petuum 0.93	CGS (SparseLDA & sample by docs)	D. P.	client-server (async)
Petuum 1.1	CGS (SparseLDA & sample by words)	M. P. (include D. P.)	ring/star topology (async)

Note: "D. P." refers to Data Parallelism. "M. P." refers to Model Parallelism.

SparseLDA algorithm with a lock-free parallel strategy to exploit local model parallelism. The design of this layer is similar to "rotateGlobal" but differs by sending documents to where the model locates rather than rotating model partitions between documents. The second layer also uses client-server architecture with asynchronous communication.

Parameter Server [27] and Petuum [28] both provide a framework to allow programming machine learning algorithms in client-server architecture with "push" and "pull" operations. Parameter Server puts the global model on servers and uses range-based "push" and "pull" operations for synchronization. These operations allow workers to update a row or a segment of parameters directly and provides a chance to batch the communication of model updates. The computation of Parameter Server's LDA implementation uses a combination of stochastic variational methods, collapsed Gibbs sampling, and distributed gradient descent. Another operation of Petuum, "schedule", allows model parallelism through scheduling model partitions to workers. Lee et al. [29] describes that the communication to fetch model data goes between clients and servers, but in the real code on GitHub [4], workers are actually directly sending data to neighbors with optimized routing.

VII. Conclusion

Through experiments on several datasets, we showed that synchronous communication methods perform better than asynchronous methods on both data parallelism and model parallelism. In data parallelism, our implementation with synchronous communication methods resulted in faster model convergence and higher model likelihood at the final iteration compared to Yahoo! LDA using asynchronous communication methods. In model parallelism, our implementation with synchronous communication methods also showed significantly lower overhead than Petuum LDA. Even though the computation speed of the first iteration is two- to threefold slower, the total execution time of our method with the same number of iterations was similar or even shorter compared with related implementations. These results prove that with optimized synchronous communication methods, we can increase the model update rate, allowing the model to converge faster, shrinking the model size, and further reducing the computation time in later iterations.

In general, despite of the implementation differences injected in performance comparison between "rtt", "lgs", Yahoo! LDA, and Petuum LDA, the advantages of synchronous communication methods are still obvious. Compared with asynchronous communication methods, synchronous communication methods can optimize routing between a set of parallel workers and maximize bandwidth utilization in point-to-point communication. Though synchronous communication methods could result in global/local waiting. However, because the word frequencies in the LDA training data is under the power-law distribution and a considerable amount of words have high frequencies, balancing the computation on all the parallel workers is feasible and the overhead of waiting is not as high as speculated. The chain reaction set off by improving the LDA model update speed amplifies the benefit of using synchronous communication methods.

In future work, we will focus on improving concurrent sampling speed on many-core systems to provide a high performance LDA implementation and apply our new communication abstractions to other machine learning algorithms facing difficulties in handling big model data.

Acknowledgment

We appreciate the system support offered by FutureSystems. We gratefully acknowledge support from NSF 1443054 SPIDAL grant.

References

[1] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *The Journal of Machine Learning Research*, vol. 3, pp. 993–102, 2003.
[2] B. Zhang, Y. Ruan, and J. Qiu, "Harp: collective communication on hadoop," in *IC2E*, 2015.
[3] "Yahoo! LDA." [Online]. Available: https://github.com/sudar/Yahoo_LDA
[4] "Petuum LDA." [Online]. Available: https://github.com/petuum/bosen/wiki/Latent-Dirichlet-Allocation
[5] P. Resnik and E. Hardist, "Gibbs sampling for the uninitiated," University of Maryland, Tech. Rep., 2010.
[6] D. Newman, A. Asuncion, P. Smyth, and M. Welling, "Distributed algorithms for topic models," *The Journal of Machine Learning Research*, vol. 10, pp. 1801–1828, 2009.
[7] J. Yuan, F. Gao, Q. Ho, W. Dai, J. Wei, X. Zheng, E. P. Xing, T.-Y. Liu, and W.-Y. Ma, "LightLDA: big topic models on modest computer clusters," in *WWW*, 2015, pp. 1351–1361.
[8] L. Yao, D. Mimno, and A. McCallum, "Efficient methods for topic model inference on streaming document collections," in *KDD*, 2009, pp. 937–946.
[9] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun, "Map-reduce for machine learning on multicore," in *NIPS*, vol. 19, 2007, p. 281.
[10] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[11] E. Chan, M. Heimlich, A. Purkayastha, and R. V. D. Geijn, "Collective communication: theory, practice, and experience," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 13, pp. 1749–1783, 2007.
[12] "fastutil." [Online]. Available: http://fastutil.di.unimi.it
[13] "FutureSystems." [Online]. Available: https://portal.futuresystems.org
[14] "clueweb." [Online]. Available: http://boston.lti.cs.cmu.edu/clueweb09/wiki/tiki-index.php?page=Dataset+Information
[15] "gutenburg." [Online]. Available: https://www.gutenberg.org
[16] "Mahout LDA." [Online]. Available: https://mahout.apache.org/users/clustering/latent-dirichlet-allocation.html
[17] "Spark LDA." [Online]. Available: http://spark.apache.org/docs/latest/mllib-clustering.html
[18] Y. Wang, X. Zhao, Z. Sun, H. Yan, L. Wang, Z. Jin, L. Wang, Y. Gao, C. Law, and J. Zeng, "Peacock: learning long-tail topic features for industrial applications," *ACM Transactions on Intelligent Systems and Technology*, vol. 6, no. 4, 2015.
[19] Y. Wang, H. Bai, M. Stanton, W.-Y. Chen, and E. Y. Chang, "PLDA: parallel latent dirichlet allocation for large-scale applications," *Algorithmic Aspects in Information and Management*, pp. 301–314, 2009.
[20] D. W. Walker and J. J. Dongarra, "MPI: a standard message passing interface," in *Supercomputer*, vol. 12, 1996, pp. 56–68.
[21] "Hadoop." [Online]. Available: http://hadoop.apache.org
[22] A. Smola and S. Narayanamurthy, "An architecture for parallel topic models," in *VLDB*, vol. 3, no. 1-2, 2010, pp. 703–710.
[23] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola, "Scalable inference in latent variable models," in *WSDM*, 2012, pp. 123–132.
[24] "Dato." [Online]. Available: https://dato.com

[25] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: distributed graph-parallel computation on natural graphs," in *OSDI*, vol. 12, 2012, p. 2.
[26] "Dato LDA." [Online]. Available: https://github.com/dato-code/PowerGraph/blob/master/toolkits/topic_modeling/topic_modeling.dox
[27] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *OSDI*, 2014, pp. 583–598.
[28] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu, "Petuum: a new platform for distributed machine learning on big data," in *KDD*, 2013.
[29] S. Lee, J. K. Kim, X. Zheng, Q. Ho, G. A. Gibson, and E. P. Xing, "On model parallelization and scheduling strategies for distributed machine learning," in *NIPS*, 2014, pp. 2834–2842.