ZenLDA: Large-Scale Topic Model Training on Distributed Data-Parallel Platform

Bo Zhao, Hucheng Zhou, Guoqiang Li, and Yihua Huang*

Abstract: Recently, topic models such as Latent Dirichlet Allocation (LDA) have been widely used in large-scale web mining. Many large-scale LDA training systems have been developed, which usually prefer a customized design from top to bottom with sophisticated synchronization support. We propose an LDA training system named ZenLDA, which follows a generalized design for the distributed data-parallel platform. The novelty of ZenLDA consists of three main aspects: (1) it converts the commonly used serial Collapsed Gibbs Sampling (CGS) inference algorithm to a Monte-Carlo Collapsed Bayesian (MCCB) estimation method, which is embarrassingly parallel; (2) it decomposes the LDA inference formula into parts that can be sampled more efficiently to reduce computation complexity; (3) it proposes a distributed LDA training framework, which represents the corpus as a directed graph with the parameters annotated as corresponding vertices and implements ZenLDA and other well-known inference methods based on Spark. Experimental results indicate that MCCB converges with accuracy similar to that of CGS, while running much faster. On top of MCCB, the ZenLDA formula decomposition achieved the fastest speed among other well-known inference methods. ZenLDA also showed good scalability when dealing with large-scale topic models on the data-parallel platform. Overall, ZenLDA could achieve comparable and even better computing performance with state-of-the-art dedicated systems.

Key words: latent Dirichlet allocation; collapsed Gibbs sampling; Monte-Carlo; graph computing; large-scale machine learning

1 Introduction

Topic models[1] provide a way to aggregate the vocabulary from a document corpus to form latent topics. In particular, Latent Dirichlet Allocation (LDA)[2] is one of the most popular topic models with rich applications in web mining, from news clustering and search topic mining to user interest profiling. Collapsed Gibbs Sampling (CGS) is the most commonly used algorithm that samples latent topics for a word occurrence (token) by integrating out the Dirichlet priors. However, training topic models with a massive corpus is challenging because of the time required and space complexity. If we consider a typical web-scale application with millions of documents and words along with thousands of topics, there would be billions of model parameters. A single machine cannot hold such a big corpus of data and big model. This motivated us to distribute the computation across multiple machines.

CGS is a serial process with high computational complexity. To achieve efficient parallel training, an
approximation approach to CGS is required with a careful tradeoff between system performance and model accuracy. Specifically, for ensuring convergence accuracy, previous studies usually resorted to sophisticated system supports to bound model staleness such as periodical update\cite{3,4}, asynchronization\cite{5,6}, and stale synchronous parallelism\cite{7-9}. Moreover, to improve training performance, they require functional supports such as mini-batch processing\cite{5,6,8} and a pipeline of model prefetching and sampling\cite{6,8}. These systems follow a customized design from top to bottom, which was built either on MPI/OpenMP\cite{10,11}, primitives\cite{3,4,12,13}, or on a parameter server\cite{7-9,14}, where each machine puts its latest update to the server and queries the server in order to retrieve recent updates from other machines. These customized approaches have to deal with sophisticated system complexities such as task scheduling, communication, synchronization, and fault tolerance. As a result, they confute the learning algorithm and system logic simultaneously, which makes debugging harder and causes loss of generality because of deep customization.

In this study, instead of customized implementations, we propose a generalized approach. The generality has three aspects. First, we propose a general Bayesian estimation method, named Monte-Carlo Collapsed Bayesian (MCCB), for learning the LDA model parameters. MCCB is a variant of CGS that can be applied easily in a parallel environment, while retaining the convergence and efficiency of CGS. Second, we propose a general graph-based topic model training framework. The framework handles data management and parameter retrieval to distribute the MCCB algorithm to a cluster of nodes. Third, we developed the entire system upon existing distributed data-parallel abstractions\cite{15-17}, which provide us underlying system-level support such as task scheduling and fault tolerance. These components decouple with each other; thus, we can perform and benefit from separate optimizations on the algorithm and system.

Spark, as a mainstream data-parallel system, has already been widely used in industrial applications. For this reason, this paper presents a concrete implementation of LDA on Spark, although the proposed algorithm framework is also suitable to different data-parallel abstractions. Developing on Spark provides us with another benefit in that our algorithm can work with others, such as in Spark MLlib\cite{18}, to easily construct an entire learning pipeline in order to be programmed and executed in the same job.

There are already two LDA systems on Spark, namely, SparkLDA\cite{19} and the official one in MLlib\cite{20} (which uses variational inference rather than CGS). However, these systems perform and scale poorly, since both of them have high computational cost. In this paper, we address the performance concern and demonstrate that such a generalized approach can achieve comparable, or even better, efficiency and scalability in comparison with state-of-the-art dedicated systems, albeit with much less engineering effort. Our contributions to both algorithm and system level optimizations are as follows:

- We convert the serial CGS algorithm to the MCCB algorithm, which can be paralleled in a fully batched and synchronized way, i.e., in each iteration, it first applies a local sampling step based on the posterior probability distribution of latent variables for all partitions, followed by synchronizing the model state at the end of the iteration.
- Based on MCCB, we decompose the LDA inference formula into parts that can be sampled more efficiently to reduce computational complexity. We call this algorithm ZenLDA.
- We propose a distributed LDA training framework, which represents the corpus as a directed graph, with parameters annotated as corresponding vertices, and then implement ZenLDA and other well-known inference methods based on Spark. This makes us utilize flexible graph partitioning approaches to achieve better performance.

We evaluated the efficiency of the proposed algorithms upon multiple datasets including a web-scale corpus. Experimental results indicate that the MCCB converged with an accuracy similar to that of CGS while running much faster. On top of the MCCB, the ZenLDA formula decomposition achieved the fastest speed among other well-known inference methods. When dealing with large-scale topic models on a cluster, ZenLDA shows good scalability. Overall, ZenLDA could achieve comparable or even better computing performance with state-of-the-art dedicated systems, such as DMTK\cite{21}.

The rest of the paper is organized as follows. Section 2 provides a brief primer on LDA, CGS, and other inference algorithms, in addition to Spark. Section 3 describes the novel LDA inference algorithm.
ZenLDA, and Section 4 describes the design of a large-scale LDA training framework. Section 5 presents the implementation and system optimization. The evaluation is described in Section 6, followed by a discussion in Section 7 and the conclusion in Section 8.

2 Background

This section describes LDA and the corresponding training algorithm, such as CGS, CVB, and SEM, and the characteristics of Apache Spark.

2.1 LDA

In LDA, each document $d$ is modeled as a mixture over $K$ latent topics, where each one is a multinomial distribution over $W$ vocabulary words. To generate a new document $d$, LDA first draws a mixing proportion $\theta_{k|d}$ from a Dirichlet prior with a parameter $\alpha$. For the $i$-th word in a document, a topic assignment $z_{di}$ is drawn as topic $k$ with probability $\theta_{k|d}$. Then, word $w_{di}$ is drawn from the $z_{di}$-th topic, with $w_{di}$ taking on value $w$ with probability $\phi_{w|k}$, where $\phi_{w|k}$ is drawn from a Dirichlet prior with parameter $\beta$. Finally, the generative process is

$$
\theta_{k|d} \sim \text{Dir}(\alpha), \phi_{w|k} \sim \text{Dir}(\beta), z_{di} \sim \theta_{k|d}, w_{di} \sim \phi_{w|z_{di}}
$$

(1)

where $\text{Dir}(\alpha)$ and $\text{Dir}(\beta)$ represent Dirichlet distributions with parameter $\alpha$ and $\beta$, respectively.

2.2 Collapsed Gibbs sampling

Given each of the observed words $w = w_{di}$, the task of the Bayesian inference for LDA is to compute the posterior distribution over the latent topic assignments $z = z_{di}$, the mixing proportions $\theta_{k|d}$, and the topics $\phi_{w|k}$. The approximate inference for LDA can be performed either by using variational methods or MCMC methods. In the MCMC context, the usual procedure is to integrate out the mixtures $\theta$ and topics $\phi$ in Formula (1), and just sample the latent variables $z$, which exhibits faster convergence. This procedure is calledCollapsed Gibbs Sampling (CGS), where the conditional probability of $z_{di}$ is computed as follows:

$$
p(z_{di} = k|z_{\neg di}, w_{di}, \alpha, \beta) \propto \frac{N_{w|k} + \beta}{N_{k} + W\beta}
$$

Here, the superscript “$\neg di$” means that the corresponding count of the current word $w_{di}$ is excluded (e.g., $N_{w|k}^{\neg di} = N_{k|d} - 1$ if $k = z_{di}$). $N_{k|d}$ denotes the number of tokens in document $d$ assigned to the topic $k$, $N_{w|k}^{\neg di}$ denotes the number of tokens of word $w$ assigned to topic $k$, and $N_{k} = \sum_{w}N_{w|k} = \sum_{d}N_{k|d}$. Note that $p$ is unnormalized.

Algorithm 1 describes the standard CGS algorithm. The processing order of line 3 and line 4 can be interchanged. There are two steps for multinomial sampling in CGS: constructing a step that computes the sampling probability of each topic $k$ (line 6), followed by a sampling step that draws a sample $z$ from the topics according to distribution $p_k$. There are four frequently used multinomial sampling approaches: Linear Search (LSearch), Binary Search (BSearch) on top of the Cumulative Distribution Function (CDF), Alias Table, and $F^+$ Tree. Table 1 lists the comparison of the time/space requirements for each of the above sampling methods.

Given documents ($D$), words ($W$), and topics ($K$), the time complexity of CGS is $O(DWK)$, and the time complexity of multinomial sampling for single token is $O(K)$. Moreover, the space requirement of the input corpus is $O(DW)$, the word-topic matrix is $O(WK)$, and the document-topic matrix is $O(DK)$. Note that their real storage could be largely reduced if a sparse data structure is used. Overall, in real web-scale applications, we need an efficient sampling algorithm and a data structure to reduce complexity.

2.3 Other related inference algorithms

Other than CGS, various approximate LDA inference techniques have been developed, such as Variational

Algorithm 1 Standard serial CGS algorithm.

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Construction time</th>
<th>Sampling time</th>
<th>Updating time</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSearch</td>
<td>$O(K)$</td>
<td>$O(K)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>BSearch on CDF</td>
<td>$O(K)$</td>
<td>$O(\log K)$</td>
<td>$O(K)$</td>
</tr>
<tr>
<td>Alias Table</td>
<td>$O(K)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>$F^+$ Tree</td>
<td>$O(K)$</td>
<td>$O(\log K)$</td>
<td>$O(\log K)$</td>
</tr>
</tbody>
</table>
Bayesian (VB)\cite{21}, Collapsed Variational Bayesian (CVB)\cite{25}, Monte-Carlo EM (MCEM)\cite{26}, Stochastic EM (SEM)\cite{27}, and so on. Here, we describe briefly some that will be modified or compared to, e.g., CVB and SEM.

In CVB, parameters $\Phi$ and $\Theta$ are marginalized out, and an MAP solution of latent topic assignments $Z$ is sought. By introducing $q(Z)$ as a variational distribution, we can obtain the lower bound $F(q(Z))$:

$$F(q(Z)) \triangleq E_{q(Z)}[\log p(X, Z|\alpha, \beta) - \log q(Z)].$$

By assuming that latent variables $Z$ are mutually independent, i.e., $q(Z) = \prod_{d,w} p(z_{d,w} | y_{d,w})$, where $y_{d,w}$ are variational parameters, we can infer the update equation as

$$\gamma_{d,w} \propto \exp \left\{ E_{q(Z|\sim d,w)} \log \left( \frac{(N_{k|d}^w + \alpha)}{N_{d}^w + K(\alpha - 1)} \frac{N_{w|k}^d + \beta}{N_{d}^w + W(\beta - 1)} \right) \right\}$$

(3)

The training procedure of CVB iteratively updates the variational parameters $\gamma_{d,w}$ by using Formula (3).

The SEM algorithm contains three steps in each iteration, namely, the Expectation step, Sampling step, and Maximization step (or E-step, S-step, and M-step, respectively), as follows:

- **E-step**: The conditional topic distribution of each token was computed as follows:

$$p(q_{d,w} = k | X, Z, \alpha, \beta) \propto \theta_{d,k} \phi_{kw}$$

(4)

- **S-step**: Draw $z_{d,w} \sim$ Multinomial$(q_{d,w})$ of each token.

- **M-step**: Update parameters using newly sampled topic assignments:

$$\theta_{d,k} = \frac{N_{k|d}^w + \alpha - 1}{N_d^w + K(\alpha - 1)}, \phi_{kw} = \frac{N_{w|k}^d + \beta - 1}{N_k^w + W(\beta - 1)}$$

(5)

### 2.4 Spark

Spark\cite{17} is a fast and general engine for large-scale data processing. It was introduced as an open-source top-level Apache project in 2013. It is growing fast, has a mature community, and has become the de-facto big data computing engine widely adopted by the industry. Spark provides the Resilient Distributed Datasets (RDDs) in-memory data abstraction. Two types of applications can benefit from RDDs, namely, iterative algorithms and interactive data mining tools. In both cases, keeping data in memory can improve performance by one order of magnitude.

#### 2.4.1 RDD abstraction

Spark improves distributed data-parallel systems such as MapReduce\cite{15}, Hadoop, and Dryad\cite{16} by providing an RDD abstraction, which is an efficient, general-purpose, and fault-tolerant abstraction for sharing data in cluster applications. Essentially, RDD represents an immutable, partitioned collection of elements that can be operated in parallel. RDDs are immutable and offered with APIs that support coarse-grained transformations, which transform RDDs and actions that return results. Faults are tolerated efficiently by using a lineage that tracks how to re-compute lost data from previous RDDs. Users can explicitly cache an RDD in memory or disk, across machines, and reuse it in successive computing. To use Spark, the developers write a driver program that implements the high-level control flow of their application and launches various RDD operations in parallel. These operations are invoked by passing a function in order to apply it on an RDD. The driver is responsible for scheduling tasks and coordinating worker execution.

#### 2.4.2 Machine learning in Spark

MLlib\cite{18} is a Machine Learning (ML) library provided by Spark. Its goal is to simplify practical ML programming by providing high-level representation Vector/Matrix/DataFrames on top of RDDs. MLlib consists of common learning algorithms and utilities as well as lower-level optimization primitives and higher-level pipeline APIs. Note that there are already two LDA implementations, with Expectation-Maximization (EM)\cite{20} on the likelihood function and variational inference based online training.

GraphX\cite{28} extends the Spark RDD with graph parallelism support by introducing a new Graph abstraction—a directed graph with properties attached to each vertex and edge. To support graph computation, GraphX exposes a set of fundamental operators (e.g., subgraph, joinVertices, and aggregateMessages) as well as an optimized variant of the Pregel API. In addition, GraphX includes a growing collection of graph algorithms and builders to simplify graph analytic tasks.

In our practice, we found that GraphX had some issues with large-scale training. First, if the attributes of vertices have complex data structures instead of simple values, the shuffling and updating of the vertices will become inefficient and even crash. Second, the APIs of GraphX are all single-threaded within each partition, which does not meet our needs. Hence, we implemented our graph-based framework directly upon...
3 Inference Algorithm for ZenLDA

In this section, we will describe a novel inference algorithm named ZenLDA. Its main novelty involves two facts: (1) It uses an MCCB algorithm to replace the serial CGS algorithm. MCCB converges to the same stationary distribution of the posterior with CGS; however, it is embarrassingly parallel and more efficient in terms of computation. (2) Furthermore, based on MCCB, we propose a different decomposition of the sampling formula from existing algorithms, which will reduce the computation cost of the distribution probabilities.

3.1 MCCB sampling method

First, we formalize MCCB and its accompanying re-sampling technique; then, we compare it with other estimation methods to explain their connections and the advantages of MCCB.

3.1.1 MCCB formalization

In the CVB algorithm, the calculation of the expectation term in Formula (3) is inefficient. In addition, the variational parameters $\gamma_{d,w,k}$ have a high storage cost. Thus, we can adopt the Monte-Carlo method to resolve these issues. Similar to the S-step in SEM, we can just calculate the distribution of $q(z_{dw})$ and draw samples from it to replace the expectation calculation of the E-step in the EM algorithm. Then, Formula (3) in CVB can be simplified to the same sampling formula as CGS (Formula (2)). Furthermore, the variational parameters can be eliminated. Thus, we obtain the MCCB inference algorithm, which resembles EM and includes two steps in each iteration:

- **S-step:** sampling topic assignment of each token $z_{dw}$ according to the posterior,
  \[ z_{dw} \sim (N_{k|dw}^{\alpha} + 1) \frac{N_{w|k}^{\beta}}{N_{k|dw}^{\alpha} + W^\beta} \]  

- **M-step:** aggregating $Z$ to update $N_{k|d}$, $N_{w|k}$, and $N_k$.

3.1.2 Re-sampling technique

Since the counters remain unchanged in MCCB during S-step, we need to exclude the current token every time when calculating the probability of the last sampled topic by using Formula (6) (i.e., $N_{k|dw}^{\alpha} = N_{k|d} - 1$ if $k = z_{dw}$), which is somewhat costly. On the other hand, in S-step, we may pre-build many alias tables for re-use, which are built based on full counters. Therefore, there is a distribution bias when using them to sample new topics for each token.

Here we propose a method called re-sampling to remedy the “subtracted by 1” problem. In other words, we build the samplers (alias table, CDF) by using full counters. Then, during sampling, if the newly sampled topic is equal to the last sampled topic, we repeat the sampling with a probability $p_{rs}$, which can be inferred as

\[ p_{rs} = 1 - \frac{p_k}{q_k} \]  

where $p_k$ is the posterior probability of State $k$ and $q_k$ is the same term with $p_k$ while calculating by using full counters (i.e., without the $\sim dw$ superscript in Formula (6)).

As an example, for the alias table built for the second topic $\alpha$-th, we can calculate the re-sampling probability by using Formula (7) to obtain (here, we ignored the changes of $N_k$ because in general $N_k \gg 1$)

\[ p_{rs} = \frac{1}{N_{w|k}}. \]

Similarly, for the 3rd part ($\frac{N_{k|dw}^{\alpha}(N_{w|k}^{\beta} + \beta)}{N_{k}^{\alpha} + W^\beta}$), it can be inferred that

\[ p_{rs} = \frac{N_{k|d} + N_{w|k} + \beta - 1}{N_{k|d} + (N_{w|k} + \beta)}. \]

In theory, the times of a distribution re-sampling can be infinite, and this follows a geometric distribution. In this situation, it can be demonstrated that the distribution calculated using full counters with re-sampling technique equals to the true posterior (here, it is Formula (6)). In practice, however, re-sampling twice at most is accurate enough to be almost the same as the true posterior.

3.1.3 Comparison with existing estimation methods

MCCB can be regarded as CGS with delayed update. In CGS, counters ($N_{k|d}$, $N_{w|k}$, $N_k$) are updated after each token sampling, which makes CGS serial. In MCCB, counters are only updated at the end of each iteration, while during S-step the counters remain unchanged. According to CVB,$^{[25]}$ we can infer that MCCB converges to the stationary posterior distribution. On
the other hand, because of the delayed update strategy, MCCB likely converges slower than CGS in the initial iterations. However, as the model update decreases during training, we can expect the convergence of MCCB to catch up to CGS when the model is largely converged. In Section 6, we will discuss various experiments that show the expected results.

Compared with MCEM or SEM, MCCB draws samples based on a true posterior distribution, while MCEM and SEM do not. It can be seen that if we let hyper-parameters \( \alpha_{MCCB} = \alpha_{SEM} - 1, \beta_{MCCB} = \beta_{SEM} - 1 \), the only difference between Formula (6) in MCCB and Formula (4) (with Formula (5) plugged in) in SEM is that the counters in MCCB are required to exclude the topic assignment of the current token. However, in SEM, the new topic distribution of a token will be somewhat biased toward its old topic, which makes the convergence of \( \theta_{dk} \) and \( \phi_{kw} \) deteriorate slightly.

### 3.2 Sampling formula decomposition of ZenLDA

In the previous section, we demonstrated that the sampling formula using MCCB is the same as the one using CGS, as shown in Formula (2). Furthermore, we can reduce the computation complexity of Formula (2). In this section, we decompose it into parts that can be calculated more efficiently. There are three major considerations regarding how to decompose better. (1) Whether the decomposed part is loop invariant or whether the change is negligible. For example, \( \frac{a\beta}{N_k + W_B} \) is loop invariant, while \( N_{k|d} N_{w|k} \) changes significantly. (2) Whether the decomposed part is sparse with respect to topic \( k \). The sparse part has less computational complexity and less memory consumption. For example, \( N_{w|k} \alpha \) is sparse since \( N_{w|k} \) is sparse, and the computational complexity is \( O(K_w) \) (the number of topics assigned for \( w \)). (3) Whether the approximation in computing topic probability does not compromise sampling accuracy. For instance, values of \( \frac{N_{k|d} N_{w|k}}{N_k + W_B} \) are relatively large, while values of \( \frac{a\beta}{N_k + W_B} \) are very small. Apparently, the approximation on the latter part would improve computing efficiency with negligible total deviation errors.

#### 3.2.1 ZenLDA decomposition

ZenLDA chooses to decompose Formula (2) to three parts, as follows (we just ignore the \( \sim dw \) superscripts for clarity):

\[
\frac{a\beta}{N_k + W_B} + \frac{N_{w|k} \alpha}{N_k + W_B} + \frac{N_{k|d} (N_{w|k} + \beta)}{N_k + W_B}.
\]

In combination with this decomposition, we adopt the word-by-word processing order and group and process all the tokens of the same word together by building an inverted index of the corpus. ZenLDA decomposition has the following benefits: (1) \( \frac{a\beta}{N_k + W_B} \) is the only dense part and is irrelevant to specific documents and words. Thus, we can compute it once and reuse it afterwards in an iteration. An alias table \( globalTable \) is built for this part, and its sampling complexity is reduced to amortized \( O(1) \). (2) \( \frac{N_{w|k} \alpha}{N_k + W_B} \) is only concerned with the current word, but not with documents. Each time, it has negligible change (with \( \frac{a}{N_k + W_B} \)); therefore, it can also be pre-computed once and reused for all tokens of the same word \( w \). Similarly, the alias table \( wordSparseTable \) is created accordingly and reduces the sampling complexity of this part to amortized \( O(1) \). Because we sample word by word, only one \( wordSparseTable \) needs to be maintained at the same time, which keeps the memory cost low. (3) \( \frac{N_{k|d} (N_{w|k} + \beta)}{N_k + W_B} \) is computed for each token with \( O(K_d) \) time complexity (the number of topics assigned for document \( d \) ). A CDF \( docSparseCDF \) is created and BSearch sampling is used. This reduces the time complexity to \( O(K_d)(construction) + O(log K_d)(sampling) \). Note that if the word occurs multiple times in the document, the CDF can be re-used, and then the construction time is amortized. Compared with the standard CGS that has \( O(K) \) complexity for each token, the ZenLDA decomposition significantly reduces the complexity to \( O(K_d) \).

#### 3.2.2 Hybrid decomposition in ZenLDA

\( K_w \) is usually larger than \( K_d \) since \( O(K_d) \) is bounded by the document length, while \( O(K_w) \) approaches \( K \) when the number of documents increases. However, this is not true for long-tail words that have fewer occurrences than \( K_d \). Thus, the corresponding word-topic array may be sparser \( (K_w < K_d) \). Therefore, we provide a hybrid sampling approach, namely, ZenLDAHybrid. For tokens (hot words) with a sparser document-topic array, we adopt the original ZenLDA decomposition; for tokens (tail words) with a sparser word-topic array, we adopt the doc-by-doc processing order and choose the decomposition as follows:

\[
\frac{a\beta}{N_k + W_B} + \frac{N_{k|d} \beta}{N_k + W_B} + \frac{N_{w|k} (N_{k|d} + \alpha)}{N_k + W_B}.
\]

This decomposition can be sampled with \( O(K_w) \) complexity. Thus, ZenLDAHybrid further reduces the sampling complexity to \( O(min(K_d, K_w)) \) for each token. However, long-tail words take up a very small percentage of a corpus, while hybrid decomposition
takes more time for additional alias table construction, which makes ZenLDAHybrid slower than ZenLDA. For simplicity, we only adopt the original decomposition as described below.

### 3.2.3 Decomposition of existing CGS algorithm

One of the trends in existing literature for improving the CGS performance is to reduce the CGS complexity by formula decomposition. Table 2 summarizes them with a detailed comparison, including SparseLDA[29], AliasLDA[23], LightLDA[8], F+LDA[24], and ZenLDA. Apart from the decomposition differences, this table also lists the differences with regard to how samplers are used, whether approximation is applied, the corresponding computing complexity of sampler construction and token sampling, and the processing order applied in the CGS step.

It is worthwhile to compare ZenLDA with LightLDA in detail, while considering that the complexity of ZenLDA is better than other alternatives, except LightLDA. First, in LightLDA, the Metropolis-Hastings (MH) steps that compute the acceptance rate of the sampled topics are required. In the MH steps, $O(1)$ complexity can be achieved only if the dense arrays or hash tables with a low load factor are used, and this will result in high memory consumption. Alternatively, if sparse vectors are used, it requires $O(\log K_w)$ and $O(\log K_d)$ complexity in order to obtain values from $N_{w|k}$ and $N_{k|d}$, respectively. Second, random access to $N_{w|k}$ and $N_{k|d}$ in the MH steps incurs high CPU cache misses, especially when $K$ is large. Consequently, it deteriorates since cyclic doc/word proposals are used and multiple MH steps are required[26]. The analysis is validated by the experiments described in Section 6. Finally, LightLDA requires an extra lookup table for each document that records the map between a token and its assigned topic. Upon sampling, LightLDA samples directly from the lookup table to simulate the alias table of $N_{k|d}$. However, it requires the corpus to be partitioned in a document-wise manner for maintaining the lookup table complete. This limits the exploration of better partitioning approaches.

### 3.3 ZenLDA algorithm

By combining the MCCB algorithm and the formula decomposition, the specific ZenLDA (MCCB) algorithm is expressed by Algorithm 2.

### 4 Large-Scale LDA Training Framework

Consider a web-scale application with billions of documents and hundreds of thousands of topics. This will result in a huge data corpus and big model size that no single machine can hold. This motivated us to distribute the computation across multiple machines. The design of a scalable and efficient web-scale LDA training is challenging and requires both data parallelism and model parallelism.

In this section, we propose a distributed LDA training framework, which represents the corpus as a directed graph, with the parameters annotated as corresponding vertices. Initially, this LDA training framework aims to scale out ZenLDA. Fortunately, its generality makes it suitable for other well-known inference methods based on CGS and MCCB.

#### 4.1 Graph-based parallelization design

### 4.1.1 Graph-based data and model representation

Instead of representing data (input corpus) and model (word-topic and document-topic) as a (sparse) matrix, we represent the data as a directed bipartite graph, which makes ZenLDAHybrid slower than ZenLDA.
Algorithm 2 ZenLDA (MCCB) sampling algorithm.

1. **Input**: Word set \( W \) and document set \( D \).
2. **Output**: Sample new topic for each token and update the model state.
3. **procedure** ZENLDA\( T \)RAINING
4.  
5.  
6.  
7.  
8.  
9.  
10.  
11.  
12.  
13.  
14.  
15.  
16.  
17.  
18.  

\[
\text{The current sampled topic } (Z_{dw}) \text{ of a token is annotated as the corresponding edge attribute. Note that the word-topic and document-topic arrays are sparse, and become increasingly sparse as the training converges. Moreover, the global state } N_k = \sum_w N_{kw} \text{ is a global variable whose value is computed by aggregating the } N_{kw} \text{ from all word vertices.}
\]

4.1.2 Partitioning approach

We achieve parallelization by partitioning the graph into multiple partitions, while the workers apply the local CGS/MCCB process to each individual partition in parallel, followed by synchronizing the model state at the end of an iteration. In this way, both data parallelism and model parallelism are achieved, where the model is also partitioned and distributed across workers.

The method to partition the corpus and model has a crucial impact on system performance and model staleness. Improper partitioning would result in load imbalance and heavy network communication overhead. In comparison with (sparse-)matrix-based representation, which can only be partitioned as a rectangle, the graph has more partitioning options. In our framework, we expect to partition the power-law graph to achieve lower communication cost and guarantee good workload balance. There exists an algorithm called Degree-Based Hashing (DBH)[311], which has been proven suitable to power-law graphs. DBH first applies the randomized hashing function to assign vertices to partitions evenly. Then, it places each edge into the partition of its source or destination vertex that has the lowest degree. DBH acquires the insight...
that locality matters for low-degree vertices, so it groups all edges with these vertices, while parallelism matters for high-degree vertices, so it favors cutting on them.

However, DBH only considers the relative size between the source degree and destination degree, without considering their absolute values. If we consider the case where both the source and destination degrees are small (smaller than a threshold value), it is not reasonable to still correspond the edge to the vertex with a lower degree, but instead, to the vertex with a higher degree. Thus, we present an algorithm called DBH+, which improves DBH with an extra degree threshold parameter. Unlike DBH, when both the degrees of the source and destination vertex of an edge are below the threshold, we place the edge into the partition of the higher-degree vertex. DBH+ works better for the corpus that has many short documents by grouping together the words in the same document, while DBH disperses them. The DBH+ algorithm is presented in Algorithm 3. The degree threshold provides performance flexibility for being tuned to fit a specific corpus. If the threshold is set to zero, DBH+ is reduced to DBH.

### 4.1.3 Synchronization approach

Here, we depict the parallel design for both CGS and MCCB to synchronize the model (across partitions), which not only eases the implementation on the data-parallel platform but also obtains similar convergence accuracy.

The CGS algorithm itself is a serial process, since there are read-write dependencies on \( N_k \). However, CGS and MCCB to synchronize the model (across partitions), which not only eases the implementation on the data-parallel platform but also obtains similar convergence accuracy.

**Algorithm 3 DBH+: Improved degree-based hash partition**

```plaintext
1. Input: Edge set \( E \); vertex set \( V \); machine set \( M \).
2. Output: Assignment \( P(e) \in M \) for each edge \( e \in E \).
3. procedure DBHPLUS
4. for each \( v \in V \) do
5. \( P(v) = \text{hash}(v) \)
6. count the degree \( d_i \) for each \( v_i \in V \) in parallel
7. for each \( e = (v_i, v_j) \in E \) do
8. if \( \max(d_i, d_j) < \text{threshold} \) then
9. \( P(e) = P(v_j) \)
10. else
11. \( P(e) = P(v_i) \)
12. else
13. if \( d_i \leq d_j \) then
14. \( P(e) = P(v_i) \)
15. else
16. \( P(e) = P(v_j) \)
```

Compared with CGS, the MCCB algorithm is embarrassingly parallel and lock-free. Thus, it is more suitable than CGS in the parallel environment and removes the partition limitation. In MCCB, a better-performed partition strategy is allowed without introducing any extra system complexity. In each training epoch, we sample locally and independently in each partition. Then, we aggregate the topic assignments to obtain new model states at the end of the epoch. Furthermore, even inside a partition, the local model update is skipped. Multi-thread techniques can be applied inside a partition without conflict and lock.

Moreover, other than easy parallelization, MCCB can also benefit from the following: (1) Alias tables are always identical to the current distribution; therefore, an extra MH step (widely used if alias table is used, such as in AliasLDA and LightLDA) is no longer required. (2) Fixed counters introduce more opportunities to compute redundancy elimination. We can pre-compute some results and re-use them to improve the calculation speed and cache hit rate.

We implemented MCCB variants for different CGS

*If a partition is document-wise partitioned, then the mini-batch is word-wise, or vice versa.*
algorithms and the evaluation indicated that MCCB achieves similar converged accuracy, especially when the corpus and the number of topics are large.

4.2 Graph-based LDA training workflow

The workflow of one iteration is illustrated in Fig. 2. It can be logically split into five steps: (1) Driver broadcasts $N_k$ to all workers. (2) Each worker sends its holding slices of model ($N_{dk}$ and $N_{wk}$) to the remote workers in need. (3) Workers apply local CGS/MCCB steps in parallel by some processing order according to the chosen algorithm. When choosing CGS, $N_{dk}$ and $N_{wk}$ are updated locally and instantly, which require synchronization or locks; in contrast, if MCCB is chosen, then $N_{dk}$ and $N_{wk}$ are updated at the epoch end, which can be lock-free. (4) Each worker sends its locally updated $N_{dk}$ and $N_{wk}$ to their responsible workers. Then, the workers aggregate them to obtain new $N_{dk}$ and $N_{wk}$. (5) The workers send all the $N_{wk}$ to the driver and the driver aggregates them to obtain new $N_k$. Almost all steps, except Step 3, have network communication, while the size of $N_k$ in Steps 1 and 5 is negligible. Thus, the network overhead is mainly caused by the shipping of $N_{dk}$ and $N_{wk}$. This will be optimized in Section 5.

5 Implementation and System Optimization

Almost all state-of-the-art large-scale LDA training systems[3–5, 8, 19, 24, 26] are designed in customized approaches that confute the learning algorithm and system logic, which makes it hard to debug and extend new algorithms on existing systems. As a comparison, our implementation was built upon a generalized data-parallel framework, namely, Spark; therefore, we can focus on core algorithm logic without worrying about system complexities and engineering efforts such as data partitioning, pipeline execution, fault tolerance, and task scheduling. First, we built a general topic model training framework on top of Spark (the training workflow in Section 4.2). Then, we implemented recent CGS algorithms such as SparseLDA, AliasLDA, LightLDA, and F+LDA, and proposed ZenLDA by using this framework. Based on the general framework, each algorithm required less than 200 lines of Scala code. Our entire implementation consisted of three layers based on Spark, as shown in Fig. 3. The code is open-source as part of the Zen project, which can be found at https://github.com/cloudml/zen/.

To obtain the best possible performance, we optimized the system from Spark to the data structure of the proposed algorithms. First, we enabled multi-threading in each partition to reduce the memory cost while keeping CPU fully utilized. Based on this, we further optimized the network shuffling overhead, data structure and computation of inference algorithms, and other trivialities. We will briefly describe these optimizations below.

5.1 Multi-threading within data partition

The data-parallel system Spark was designed to process one partition per core, and all the processing partitions must be loaded in memory. Thus, “Out of memory” often occurs if many partitions (16–32 cores per machine) are loaded concurrently. The common solution is to reduce the size of the partition. However, this does not work well here, because the partitioning power-law graph to more parts would increase network shuffling dramatically. Therefore, to resolve the memory bottleneck while achieving better performance, our strategy was to leave each partition be as large as possible, load fewer partitions (less than cores) at one time, and enable multi-threading in each partition to fully utilize the CPU cores.

Since GraphX does not support multi-threading within the partition (and other issues depicted in Section 2.4), we could not use GraphX directly. We
implemented our framework mimicking GraphX on top of Spark RDD, albeit with complete improvements from basic data structure to APIs. For example, in GraphX, there was an array in each EdgeRDD partition that recorded the source vertex of each edge. If we consider that the edges in one EdgeRDD partition are sorted in source vertex order, then this array would have large redundancy and can be largely reduced by just recording the starting offset for each source vertex. Moreover, this optimization not only saves the memory of EdgeRDD (reduced about 1/3 size) but also enables multi-threading facilities. Each thread will read the range of a source vertex and process all edges within the range. Threads obtain source vertices in a work-stealing mode to achieve good load balance. In addition, the APIs of GraphX are serial and designed for partition-wise use. Thus, we have to abandon them and re-implement the multi-threaded version of the needed APIs (and even new ones). If the algorithm is CGS, it requires synchronization between threads; otherwise, if the algorithm is MCCB, the threads are totally lock-free. Note that with multi-threading techniques enabled, the background processing of the Spark framework is free. With multi-threading facilities, each thread will read the range of a source vertex and process all edges within the range. Threads obtain source vertices in a work-stealing mode to achieve good load balance. In addition, the APIs of GraphX are serial and designed for partition-wise use. Thus, we have to abandon them and re-implement the multi-threaded version of the needed APIs (and even new ones). If the algorithm is CGS, it requires synchronization between threads; otherwise, if the algorithm is MCCB, the threads are totally lock-free. Note that with multi-threading techniques enabled, the background processing of the Spark framework is free.

5.2 Efficient network communication

As shown in Section 4.2, the model \((N_{dk}, N_{wk})\) shuffling cost in Steps 2 and 4 is a critical performance factor, especially when the topic and partition numbers are large. The proposed DBH+ partitioning approach and the Kryo serialization library\([32]\) provided in Spark have already reduced the shuffling cost significantly. Moreover, we adopt two more optimizations to improve network performance further.

5.2.1 Model compression

We compress \(N_{dk}\) and \(N_{wk}\) by using JavaFastPFOR\([33]\), a special compression library special for integers, which achieves a much higher compression rate and runs much faster than general compressors, and thus reduces the storage. Furthermore, we encapsulate SparseVector and DenseVector as CompressedVector. We store and shuffle the vectors in compressed form, and decompress them during sampling with negligible cost. This optimization reduces the cost only in Step 2, because we need the aggregation operation among vectors in Step 4, and thus we cannot compress them.

5.2.2 Delta aggregation

In Step 4, we reduce network I/O further. With the insight that in later iterations, a high proportion of tokens converges without topic change, we present delta aggregation that only changed locally aggregated tokens. Thus, network I/O is largely reduced as the model converges. This requires retaining both the old and the new sampled topic for each token, thus it doubles the edge attribute size.

5.3 Inference algorithm optimization

In addition to the optimization for the data-parallel platform, we also adopt many techniques to improve the speed of inference procedures such as sampling and counting.

5.3.1 Sparse-dense-aware data structure

Inherently sparse exists in word-topic arrays and document-topic arrays; thus, we store them by DenseVector and SparseVector provided in the Breeze\([34]\) library. \(N_{dk}\) is always represented as SparseVector, while \(N_{wk}\) is represented as SparseVector by default. However, once its active size becomes larger than \(K/4\), it will be converted to DenseVector. SparseVector is more memory efficient if it has large sparsity; however, its indexing cost that happens at the third part of the ZenLDA decomposition, when accessing \(N_{wk}\), is increased from \(O(1)\) to \(O(\log K_w)\). To eliminate the indexing cost, we expand the word-topic vector of the word to a dense one before processing its tokens.

5.3.2 Alias table improvement

The alias table is well-known for its low sampling complexity. However, its construction time is costly (high complexity coefficient) and needs three random generated numbers to draw a sample, as the construction method presented in AliasLDA\([23]\). To reduce the construction cost, we refine the algorithm that inserts topics and threshold directly into the corresponding bin in the alias table; thus, two queues of topic probabilities in the original algorithm are eliminated. Moreover, we scale the threshold of each bin by multiplying the sum of probabilities. This allows us to reuse the random number generated from upper-level samplers in order to flip the bias coin in a bin, which now needs just one random generated number (to choose the bin) to draw a sample.

5.3.3 Redundant computation elimination

The MCCB algorithm with delayed update exposes
many redundant computations in the S-step. For instance, \( \frac{1}{N_k + W} \) will be calculated repeatedly during the entire iteration. Thus, we can pre-compute it, and then re-use the result. For each hot word \( w \), we can also pre-compute \( \frac{N_w + \beta}{N_k + W} \) for each topic \( k \), which will be reused in the third part of the ZenLDA decomposition. This benefits CPU cache usage and reduces memory footprint.

5.4 Scala-related optimization

Scala is a programming language based on JVM. Careful programming to avoid boxing/unboxing and generation of closures can reduce the CPU cost for the purpose of making Scala as fast as possible. Many advanced features such as for-loops, anonymous functions, and generics should be avoided in the main execution loop. To reduce JVM garbage collection, arrays should not be created on the fly, but rather be pre-allocated and the space be re-used.

6 Evaluation

The evaluations aim to demonstrate the effectiveness and efficiency of ZenLDA in comparison with other CGS algorithms; the effectiveness and efficiency of MCCB in comparison with the corresponding CGS version; and the scalability of ZenLDA that varies the topic number, the dataset size, and number of machines; and the speed comparison between ZenLDA and state-of-the-art LDA training systems.

6.1 Evaluation settings

6.1.1 Datasets

We used four different datasets, including a small sized NYTimes (520 MB), a medium sized PubMed (3.8 GB), a large one-month web chunk data indexed by Bing News (BingWeb1Mon, 17 GB), and a super large-scale Bing data (BingWeb320G). They were all pre-processed and saved in libSVM format. The detailed information is listed in Table 3.

6.1.2 Cluster configuration

We had three Spark clusters at different scales. The smallest one consisted of eight homogeneous computing nodes connected via 1 Gbps Ethernet. Each node had 12 2.10-GHz Intel(R) Xeon(R) E5-2620 cores with multi-threading enabled. The driver was configured with 4 GB memory and the eight workers were configured with 40 GB memory. The experiments with NYTimes and PubMed were conducted in this cluster. Note that except the serial CGS algorithm running on a single machine with a single thread, all other algorithms run in a distributed environment, in which NYTimes was executed in a single machine with three partitions that had eight threads each. PubMed was executed by using all eight machines, which were split into 24 partitions with eight threads each. The medium-scale cluster had 10 homogeneous computing nodes, which were connected via a 40-Gbps Infiniband network and each node consisted of 16 2.40-GHz Intel(R) Xeon(R) CPU E5-2665 cores. The driver was configured with 5 GB memory and the 10 workers were configured with 100 GB memory. BingWeb1Mon was evaluated in this medium cluster with 40 partitions, where each partition had four threads. The largest Spark cluster was deployed on a multi-tenancy cluster managed by Yarn\(^{35}\), where the resource was not always guaranteed and an executor was configured to have 20 GB memory and 10 cores. The nodes of the cluster were located in data centers around the world and were connected via VPN. The scalability experiments against BingWeb320G were conducted in this cluster.

6.2 Evaluation of CGS algorithms

First, we compared different CGS algorithms to demonstrate the convergence and efficiency of ZenLDA in comparison with existing CGS algorithms. In this evaluation, we did not use the full-fledged ZenLDA algorithm proposed in Section 3, but rather the CGS version of ZenLDA, which is the ZenLDA decomposition on top of CGS. This intended to compare ZenLDA with other algorithms, with regard to fair play, by just reducing the computational complexity.

We implemented SparseLDA, AliasLDA, LightLDA, F+LDA, and the CGS version of ZenLDA by using the same general framework proposed in Sections 4 and 5, based on Spark. All implementations were with asymmetric priors\(^{36}\) and applied the same optimization (Section 5). Each proposal distribution in AliasLDA and LightLDA only had one MH step. The more MH steps, the more accuracy improvement...
with performance slowdown. We cannot compare to SparkLDA since it is not open-source; however, we believe that ZenLDA will win since SparkLDA uses the standard inference algorithm, which is computationally expensive. In addition, we did not compare to the EM-based LDA implementation in MLlib, because it is much slower and cannot even finish the first iteration against the PubMed dataset, while exceptions also occurred. The comparison was made with NYTimes against PubMed datasets with 1000 and 10,000 topics, respectively. All experiments had the same $\alpha$ and $\beta$ as 0.01, and were executed with 100 iterations.

Figure 4 illustrates the comparison on model convergence (log-likelihood). SerialCGS represents the accuracy baseline that was evaluated using the standard serial CGS algorithm. Among different CGS algorithms, SparseLDA and F+LDA had the best convergence of distributed algorithms since no local approximation was applied. AliasLDA and the CGS version of ZenLDA also shared similar accuracy (hard to distinguish), since the proposal distribution was almost the same as the original distribution. LightLDA performed worst, with respect to log-likelihood, and even worse as the number of topics increased. This may be due to the proposal distribution used in LightLDA being far from the true probability.

Figure 5 depicts the execution time ($y$-axis) of each iteration ($x$-axis). The spikes in Fig. 5 stem from GC in JVM and network fluctuation. We do not include SparseLDA in this figure, since it is relatively slow, e.g., the average time per iteration is 204.8 s on the smallest data (NYTimes) with the smallest topics ($K = 1000$). The evaluation validated the performance order as SparseLDA < AliasLDA < F+LDA < ZenLDA. ZenLDA was the best performer and the speedup was retained almost the same among different datasets and different topic numbers.

The performance of LightLDA was somewhat surprising considering that it only had $O(1)$ complexity. With more data and more topics, LightLDA performed even worse. This was largely due to the slow convergence of LightLDA with the denser model, which resulted in larger network I/O. Moreover, there were two implementation related reasons: (1) We implemented the alias table rather than the look-up table for document proposal because the data were not partitioned in a document-wise manner; (2) As we discussed in Section 3.2.3, since the sparse representation of $N_{k_id}$ and $N_{w_jk}$ was used, the MH steps in LightLDA would be costly with $O(\max(K_w, K_d))$. We did not represent them as hash tables since this would require more memory space and would result in more cache misses.

![Fig. 4 Log-likelihood comparison among different CGS algorithms. Note that the curves of SparseLDA, F+LDA, AliasLDA, and ZenLDA-CGS are almost the same and some of them get covered.](image-url)
6.3 Evaluation of MCCB compared with CGS

For ZenLDA, LightLDA, and AliasLDA, we also implemented MCCB variants of corresponding CGS algorithms. SparseLDA was ignored because it is relatively slow, and F+LDA was skipped since F+ tree is designed for instant update and it is not suitable to a delayed update.

The model convergence comparisons are shown in Fig. 6. Note that the curves of ZenLDA-CGS and ZenLDA-MCCB are almost covered by the curves of AliasLDA and AliasLDA-MCCB. The evaluation indicates that MCCB converges with an accuracy similar to that of CGS. In comparison with the CGS version, the corresponding MCCB variant converged a bit slower at first, but it finally caught up after 60–80 iterations. LightLDA behaved differently because its MCCB variant converged much slower at first, but caught up and even outperformed the CGS version after approximately 60 iterations.

Figure 7 depicts the execution time comparisons. We can see that the MCCB significantly outperformed the corresponding CGS version, since it was totally lock-free, MH steps were avoided in AliasLDA-MCCB and ZenLDA-MCCB (LightLDA-MCCB still had MH steps), and more pre-computing could be applied. It was also indicated that in all experiments the execution time decreased as the model became sparser and until it converged. The first several iterations were always the most time-consuming parts.

6.4 Scalability evaluation

6.4.1 Performance by increasing topic number (scaling-up)

We evaluated the performance with varying topic number. The experiment was conducted against BingWeb1Mon with 1000, 10 000, and 100 000 topics, respectively. The larger topic number had larger shuffling cost that was gradually reduced until the model converged. Their training time for the first 60 iterations is shown in Fig. 8. The stable average time per iteration (average time per iteration after 30-th) was only increased from 34 s to 44 s when the topic number increased from 1000 to 10 000. Even with 100 times more topics (100 000), the stable average time per iteration was only increased to 69 s. Thus, ZenLDA is very scalable when the topic number increases.

6.4.2 Performance by using more nodes (scaling-out and fault tolerance)

The scalability experiments were conducted only for ZenLDA against a huge Bing-Web320G dataset with 10 000 topics and were run on the Bing multi-tenancy data center. Each partition was executed in an executor.
Fig. 6 Log-likelihood comparison between CGS and MCCB. Note that the curves of ZenLDA-CGS/MCCB were almost covered by AliasLDA-CGS/MCCB.

Fig. 7 Time cost comparison between CGS and MCCB.

(container) with 10 threads. Figure 9 indicates that ZenLDA scaled pretty well. With more executors added (240 vs. 120, 360 vs. 240), the performance could be fairly improved, although with less linear speedup because the network I/O became larger (the more executors, the more shuffling data). Note that during the experiments, some failures occurred because the tenanted machines were potentially taken away at random. ZenLDA needs to mark checkpoints every several iterations and can revive from last saved
checkpoints automatically if some machines are lost. The fault or straggler tolerance supports in Spark make ZenLDA more promising in production requiring large scalability, easy deployment, and execution in a shared environment.

6.5 Comparison with state-of-the-art system

We compared the MCCB version of ZenLDA (the best performer in Spark) with the DMTK system on the medium cluster. DMTK is considered as the state-of-the-art LDA training system that implements the LightLDA ($O(1)$ complexity) algorithm on top of the Parameter Server with approximately 3000 lines of native C++ client code. DMTK also supports asynchronization with sophisticated design to hide the network communication via pipeline execution and prefetching. DMTK reports log-likelihood every 5 iterations.

The evaluation was made for all datasets except BingWeb320G, and the performance of total training time is listed in Table 4. It shows that ZenLDA achieves similar performance as DMTK for the NYTimes and PubMed datasets and is even 27.6% faster (95.5 s vs. 121.9 s) in a larger BingWeb1Mon dataset with 100,000 topics. This is because the ratio of fixed overhead in ZenLDA is largely reduced in the large dataset.

The running time of each iteration training on BingWeb1Mon is depicted in Fig. 8. As can be seen, ZenLDA has much better performance in 10–50 iterations, and DMTK gradually catches up as the model becomes sparser; thus, the communication cost is reduced. In Fig. 4, it is shown that LightLDA converged much worse than ZenLDA at first; the worse convergence would also result in larger communication cost, which would lead to poor performance. The performance difference between the two LightLDA implementations (DMTK vs. Spark) indicates the language cost (C++ vs. Scala) and framework cost (Parameter Server vs. Spark). We believe that ZenLDA can be sped up further if it is implemented in C++; however, this will introduce more system complexities and engineering effort.

7 Discussion and Future Work

This section describes several points that are out of this paper’s scope, but that could be considered in future work.

7.1 Optimization of training procedure

MCCB is a special MCMC method, which usually has the following issues: (1) Model parameters are initialized randomly, which makes the initial model dense. Hence, in the first several iterations, it usually has high computation and storage cost, which can cause performance and scalability bottleneck. (2) After many iterations, the parameters become sparser as the training progresses and the model tend to converge to the stationary distribution. In these iterations most of the samples remain unchanged and most computation is a waste of time. Therefore, we should optimize the beginning iterations and boost largely converged iterations with some further optimizations.

7.2 Auto-tuning of system parameter

In our framework, the number of partitions and threads should be set appropriately to obtain the best performance. However, there is no general best value
and it all depends on the cluster’s hardware. These parameters can only be tuned based on experiences and testing. We are in the process of establishing a strategy to set these optimal numbers automatically.

8 Conclusion

In this paper, we presented the ZenLDA algorithm and techniques to perform efficient and scalable LDA training on the distributed data-parallel platform. The experimental results demonstrated that ZenLDA can achieve comparable and even better computing performance with the state-of-the-art dedicated systems. ZenLDA also shows good scalability when dealing with large-scale topic models. The proposed algorithm and framework are general and useful to other system implementations. By combining the algorithm with system improvements, we demonstrated that developing a distributed machine learning system should combine indispensable innovations from both the algorithm and system sides. In addition, the distributed data-parallel abstraction (especially graph abstraction) is not only feasible and beneficial but also efficient and scalable. We will continue to develop this methodology and will test it on larger-scale machine learning models in the future.

Acknowledgment

This work was initially studied during the internship at Microsoft Research Asia (MSRA). We thank MSRA for the support for this work. This work was partially supported by the National Natural Science Foundation of China (No. 61572250) and the Science and Technology Program of Jiangsu Province (No. BE2017155).

References

Bo Zhao is a PhD candidate in Department of Computer Science and Technology, Nanjing University. His main research interests include text mining and big data parallel processing. He obtained the bachelor degree from Northwest Agriculture and Forestry University in 2011.

Hucheng Zhou is working at Microsoft Research Asia. He obtained PhD degree from Tsinghua University in 2011. His research interests include large-scale learning system, data-parallel computing, big data, machine learning, mobile computing, program analysis, compiler optimization, computer architecture, and tool development.

Guoqiang Li is a senior engineer for Russell Department at Central Software Institute of HUAWEI 2012 Laboratory. He is mainly responsible for research and development of big data processing platform. He obtained the bachelor degree from Jiujiang University in 2012.

Yihua Huang is a professor of the Department of Computer Science and Technology, Nanjing University. He received the PhD degree in computer science from Nanjing University in 1997. He is a senior member of China Computer Federation. His research interests include Web data mining, text mining and semantic analysis, RDF database, and big data parallel processing.