Efficient and Scalable Topic Model Training on Distributed Data-Parallel Platform

Bo Zhao[1,2], Hucheng Zhou[1], Guoqiang Li[3] and Yihua Huang[2]
[1]*Microsoft Research* [2]*Nanjing University* [3]Huawei Inc.

Submission Type: Research

May 28, 2016

Abstract

Distributed Collapsed Gibbs Sampling (CGS) in Latent Dirichlet Allocation (LDA) training usually prefers a "customized" design with sophisticated asynchronization support. However, with both algorithm level innovation and system level optimizations, we demonstrate that the "generalized" design on distributed data-parallel platform can even outperform the dedicated designs. We first present a novel CGS sampling algorithm, ZenLDA, that has different formula decomposition with different performance-accuracy tradeoff with other CGS algorithms. With respect to parallelization, we convert the serial CGS algorithm to Monte Carlo Expectation-Maximization (MCEM) algorithm thus can be parallelized in a fully batched and synchronized way. To push the performance to the limit, we also present two approximations, sparse model initialization and "converged" token exclusion, as well as several system level optimizations. Training corpus is represented as a directed graph and model parameters are annotated as the corresponding vertex attributes, thus we implemented ZenLDA and other well-known CGS algorithms on GraphX in Spark, and it has been deployed and daily used in production. We evaluated the efficiency of presented techniques against multiple datesets including web-scale corpus. Experimental results indicate that MCEM variant achieves much faster than CGS algorithms but still converges with similar accuracy, and ZenLDA is the best performer. When compared with state-of-art systems, ZenLDA achieves comparable (even better) performance with similar accuracy. Besides, ZenLDA demonstrates good scalability when dealing with large topics and huge corpus.

1 Introduction

Topic models provide a way to aggregate vocabulary from a document corpus to form latent "topics". In particular, Latent Dirichlet Allocation (LDA) [8] is one of the most popular models [8, 17] that has rich applications in web mining, from News clustering, search topics mining even to user interests profiling. Collapsed Gibbs Sampling (CGS) is the most commonly used algorithm that samples the latent topics for a word occurrence (token) by integrating out the Dirichlet priors. However, the training with massive corpus is challenging because of high time and space complexity. Consider a typical web-scale application with millions of documents and words, as well as thousands of topics, there would be billions (even trillions) of model parameters. No single machine can hold such Big corpus data nor Big model size, which motivates a scalable and efficient way of distributing the computation across multiple machines.

Since CGS itself is a serial process, the efficient parallelized training requires a certain degree of staleness (approximation) with careful tradeoff between system performance and model accuracy. For instance, to ensure the converged accuracy, existing work usually resorted to sophisticated system supports to bound the model staleness such as periodical update [24, 30], asynchronization [5, 22], and stale synchronous parallelism [16, 36, 33]. Besides, to improve the training performance, they required system supports such as mini-batch processing [5, 22, 36] and pipeline processing of data prefetching and sampling process [22, 36]. On the one hand, however, these requirements increase the system complexity that leads to a `customized` design. They were build either on MPI [1]/OpenMP [7] primitives [31, 28, 30, 24], or on a parameter server [21, 33, 16] abstraction that each machine put its latest update to server and query the server to retrieve recent updates from other machines. On the other hand, this excludes system choice on `generalized` distributed data-parallel platform such as Spark, since shuffle-based synchronization in data-parallel execution is considered to have poor performance due to straggler (compared with asynchronization in parameter server) and the full batched execution would introduce excessive model staleness with possible accuracy loss.

However, those `customized` approaches conflate

the learning algorithm and system logic together, which makes it hard to debug and extend new algorithms on old systems. They repeatedly address the same system challenges, and lose generality due to deep customization. Up to now, there is still no general system that supports all different CGS algorithms. Instead, we prefer the `generalized` approach with a distributed data-parallel abstraction [12, 18, 37] between machine learning algorithm and underlying system. Therefore, both algorithm advancement and system improvement can benefit the training performance. Besides, system complexities such as scheduling, communication and fault tolerance are hidden. Consider that data-parallel system such as Spark has already been widely adopted in industry, this provides another benefit that entire learning pipeline, from feature engineering to model training, can be programmed in one job and executed in the same framework. Hadoop Mahout [6] and Spark MLlib [37, 23] have validated such generalized approach. There are already two LDA systems on Spark, SparkLDA [25] (no source code available) and the official one in MLlib [9] (Expectation-Maximization (EM) rather than CGS algorithm). However, they are considered to be performed and scaled poorly (10˜100X worse) or even not scalable at all. In this paper, we address the performance concern and demonstrate that such generalized approach can still achieve comparable or even better efficiency and scalability but with much simpler engineer efforts, with the combined contributions from both algorithm level innovation and system level optimizations:

- We first present a novel CGS sampling algorithm, ZenLDA, that balances the time complexity, model accuracy and parallelization flexibility.
- We convert serial CGS algorithm to Monte Carlo EM (MCEM) algorithm that can be parallelized in a fully batched and synchronized way, i.e., in each iteration it first applies local (CGS) sampling step for all partitions, followed by synchronizing the model state at the iteration end.
- To push the performance into limit, we also present two approximations, sparse model initialization and "converged" token exclusion, as well as several system level optimizations.
- We have implemented ZenLDA and other well-known CGS algorithms on GraphX/Spark, where input corpus is represented as a directed graph and model parameters are annotated as the vertex attributes. This makes us utilize flexible graph partitioning approaches to achieve better performance.

We evaluated the efficiency of presented techniques against multiple datesets including web-scale corpus. Experimental results indicate that MCEM variant achieves much faster than CGS algorithms but still converges with similar accuracy, and ZenLDA is the best performer. When compared with state-of-art systems such as DMTK [26], ZenLDA achieves comparable (even better) performance with DMTK but better accuracy than DMTK. Besides, ZenLDA demonstrates good scalability when dealing with large topic number and huge corpus.

2 Background

This section describes LDA and the corresponding CGS training algorithm. Due to page limitation, we skipped the description of Apache Spark [37] and its graph computing library GraphX [14].

2.1 LDA

In LDA, each of D documents is modeled as a mixture over K latent topics, each being a multi-nomial distribution over W vocabulary words. In order to generate a new document d, LDA first draws a mixing proportion $\theta_{k|d}$ from a Dirichlet prior with parameter α. For the wth word in the document, a topic assignment z_{wd} is drawn as topic k with probability $\theta_{k|d}$. Then word x_{dw} is drawn from the z_{dw}th (kth) topic, with x_{dw} taking on value w with probability $\phi_{w|k}$, where $\phi_{w|k}$ is drawn from a Dirichlet prior with parameter β. Finally, the generative process is:

$$\theta_{k|d} \sim Dir(\alpha), \phi_{w|k} \sim Dir(\beta), z_{dw} \sim \theta_{k|d}, x_{dw} \sim \phi_{w|z_{dw}} \quad (1)$$

where $Dir(\alpha)$ and $Dir(\beta)$ represent Dirichlet distribution.

2.2 Collapsed Gibbs Sampling Algorithm

Given the observed words $x = x_{dw}$, the task of Bayesian inference for LDA is to compute the posterior distribution over the latent topic assignments $z = z_{dw}$, the mixing proportions $\theta_{k|d}$ and the topics $\phi_{w|k}$. Approximate inference for LDA can be performed either using variational methods [8] or Markov chain Monte Carlo (MCMC) methods [15]. In the MCMC context, the usual procedure is to integrate out the mixtures θ and topics ϕ in Formula 1 and just sample the latent variables z, which exhibits fast convergence. This procedure is called Collapsed Gibbs Sampling (CGS), where the conditional probability of z_{dw} is computed as follows:

$$p(z_{dw} = k | z^{\neg dw}, x_{dw}, \alpha, \beta) \propto (N_{k|d}^{\neg dw} + \alpha) \frac{N_{w|k}^{\neg dw} + \beta}{N_k^{\neg dw} + W\beta} \quad (2)$$

where the superscript $\neg dw$ means the corresponding topic sampled last time is excluded (e.g., $N_{k|d}^{\neg dw} = N_{k|d} - 1$ if $k = z_{dw}$), $N_{k|d}$ denotes the number of tokens in document d assigned to topic k, $N_{w|k}$ 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 [1]. There are two steps for multinomial sampling in CGS,

[1] The processing order of line 3 and line 4 can be interchanged.

Algorithm 1 Serial standard CGS algorithm.

```
1: procedure STANDARDCGS
2:     for each epoch e do
3:         for each document d do
4:             for each word w do
5:                 for each topic k do
6:                     p(k) = (N_{k|d}^{¬dw} + α) (N_{w|k}^{¬dw} + β) / (N_k^{¬dw} + Wβ)
7:                 t = TopicSampling(p(k))
8:                 update N_{t|d}, N_{t|k} and N_k accordingly
```

`constructing` step that computes the sampling probability of each topic k (line 6), followed by `sampling` step that draws a sample z from topics such that $P(z = k) \sim p_k$. There are four frequently used multinomial sampling approaches [35], `linear search`, `binary search`, `alias table` [20, 35] and `F+ tree`. Table 1 lists the comparison of the time/space requirements of each of the above sampling methods.

	LSearch	BSearch	Alias Table	F+ Tree
Construct time	$O(K)$	$O(K)$	$O(K)$	$O(K)$
Sample time	$O(K)$	$O(\log K)$	$O(1)$	$O(\log K)$
Update time	$O(1)$	$O(K)$	$O(K)$	$O(\log K)$

Table 1: Comparison of multinomial samplers. CDF means cumulative distribution function.

Given documents (D), words (W) and topics (K), the time complexity of CGS is $O(D * W * K)$ that the time complexity of multinomial sampling for single token is $O(K)$. And the space requirement of input corpus is $O(D * W)$, word-topic matrix is $O(W * K)$, and document-topic matrix is $(D * K)$. Note that all of them are sparse thus the real storage could be largely reduced if sparse data structure is used. Consider real web-scale application, we need an efficient sampling algorithm and data structure to reduce the complexity.

3 CGS algorithm

3.1 ZenLDA

We first describe ZenLDA, a novel serial sampling algorithm that has different decomposition of Formula 2 with existing CGS algorithms. There are three major considerations: 1) whether the decomposed part is loop invariant or with negligible change? For example, $\frac{\alpha*\beta}{N_k+W\beta}$ is loop invariant while $N_{k|d} * N_{w|k}$ changes significantly. 2) whether the decomposed part is sparse with respect to topic k? Sparse part has less computing complexity as well as less memory consumption. For example, $N_{w|k} * \alpha$ is sparse since $N_{w|k}$ is sparse, and the computing complexity is $O(K_w)$ (the number of topics assigned for w). 3) whether or not the approximation in computing topic probability does not compromise the sampling accuracy? Apparently, the approximation on smaller value part would have less deviation errors in total. It is thus unnecessary to compute the less important part every time. For instance, $N_{k|d} * N_{w|k}$ has largest value while $\alpha * \beta$ is the smallest.

ZenLDA decomposition. ZenLDA chooses a different decomposition with $\frac{\alpha*\beta}{N_k+W\beta} + \frac{N_{w|k}*\alpha}{N_k+W\beta} + \frac{N_{k|d}*(N_{w|k}+\beta)}{N_k+W\beta}$, which has the following benefits compared with other approaches: 1) $\frac{\alpha*\beta}{N_k+W\beta}$ is invariant thus only computed once and reused afterwards in an iteration. And an alias table, *gTable*, is created accordingly, thus $O(1)$ sampling complexity is achieved. 2) $\frac{N_{w|k}*\alpha}{N_k+W\beta}$ has negligible change (with $\frac{\alpha}{N_k+W\beta}$) each time, so it is also pre-computed once and reused for all tokens of the same word (w). Similarly, alias table (*wSparse*) is created accordingly. Lifecycle of *wSparse* is reduced if word-by-word process order is adopted that all tokens of the same word are grouped and processed together. Note that alias table construction cost has been amortized. 3) $\frac{N_{k|d}*(N_{w|k}+\beta)}{N_k+W\beta}$ is computed for each token with $O(K_d)$ time complexity (the number of topics assigned for document d). And a cumulative distribution function (CDF) is created and the corresponding sampling complexity is $O(\log K_d)$. Note that it is only computed once for multiple occurrences with the same (word, document) pair.

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 to K when the number of documents increases. However, this is not true for long-tail words that have less occurrences than document length, thus the corresponding word-topic array may be more sparse ($K_w < K_d$). Therefore, we further provide a hybrid sampling approach, `ZenLDAHybrid` that: 1). for tokens with more sparse document-topic array, we adopt decomposition as $\frac{N_{w|k}*\alpha}{N_k+W\beta} + \frac{N_{k|d}*(N_{w|k}+\beta)}{N_k+W\beta}$ (with $O(K_d)$ complexity); 2). for tokens with more sparse word-topic array, we adopt decomposition as $\frac{N_{k|d}*\beta}{N_k+W\beta} + \frac{N_{w|k}*(N_{k|d}+\alpha)}{N_k+W\beta}$ (with $O(K_w)$ complexity).

Probability Approximation. Beyond formula decomposition, ZenLDA also use some approximations to further improve sampling efficiency while barely affect the convergence. 1) When Alias method is used as proposal distribution, such as in AliasLDA [20] and LightLDA [36], the Metropolis-Hastings (MH) step is need because the proposal departs from the true distribution. However, ZenLDA always accepts the newly sampled topic [2]. 2) Recall that when computing $p(z_{dw} = k)$ in CGS (Formula 2), the counters should be subtracted by one for topic sampled last time. However, we do not directly subtract one since we reused the second decomposed term

[2] Accept probability ($\frac{p_j*q_i}{p_i*q_j}$) in MH steps always equals to 1 if we only update the model at the iteration end (Section 4.1)

Algorithm 2 Serial sampling algorithm in ZenLDA.

procedure ZENCGSTRAINING
  for each epoch e **do**
    for each topic $k \in K$ **do**
      $gDense \leftarrow \frac{\alpha*\beta}{N_k+W\beta}$
    $gTable \leftarrow createAliasTable(gDense)$
    for each word $w \in W$ **do**
      for each topic $k \in K_w$ **do**
        $wSparse \leftarrow \frac{N_{w|k}*\alpha}{N_k+W\beta}$
      $wTable \leftarrow createAliasTable(wSparse)$
      for each token $t = (w, d_i) \in E, d_i \in D$ **do**
        for each topic $k \in K_d$ **do**
          $dSparse \leftarrow \frac{N_{k|d}*(N_{w|k}+\beta)}{N_k+W\beta}$
        $dCDF \leftarrow createCDF(dSparse)$
        $z_t \leftarrow sample(gTable, wTable, dCDF)$ with resampling

($\frac{N_{w|k}*\alpha}{N_k+W\beta}$). We remedy this by resampling, i.e., if the newly sampled topic is equal to the topic sampled last time we redo sampling with a probability $\frac{1}{N_{w|k}}$ (we ignored the changes of N_k because it is far larger than 1). Similarly, for the 3rd term ($\frac{N_{k|d}*(N_{w|k}+\beta)}{N_k+W\beta}$), the resampling probability is $\frac{N_{k|d}+N_{w|k}+\beta-1}{N_{k|d}*(N_{w|k}+\beta)}$.

ZenLDA algorithm. The specific serial ZenLDA algorithm is described in Algorithm 2. We skipped the algorithm for ZenLDAHybrid since it is a natural extension. Compared with standard CGS that has $O(K)$ complexity, ZenLDA significantly reduces the complexity into $O(min(K_d, K_w))$.

3.2 Related work in CGS algorithm

Two trends exist in literature to improve the CGS performance. One of the trend aims to reduce the CGS complexity. Table 2 summarizes them with the detailed comparison, including SparseLDA [34], AliasLDA [20], LightLDA [36] and F+LDA [35]. Besides the difference in decomposition, this table also lists the difference on which `sampler` is used, whether it is `fresh` that the formula value is freshly computed rather than reuses the old value, whether `approximation` is applied, the corresponding `computing complexity` if computing is needed, the `sampling complexity`, and the `process order` applied in CGS step.

It is worthy to compare ZenLDA with LightLDA in details, consider that the complexity of ZenLDA is better than other alternatives except LightLDA. First, LightLDA needs an extra lookup table for each document that records the map between a token and its assigned topic. Instead of reading $N_{k|d}$ directly, LightLDA samples the lookup table to simulate $N_{k|d}$, thus the complexity is reduced from $O(K_d)$ to $O(1)$. However, it requires data is partitioned in a document-wise way, otherwise the lookup table would be incomplete that some tokens of this document are at different partitions. This limits the exploration of better partition approaches . Second, MH steps are needed, which will compute the acceptance rate of the sampled topic, $O(1)$ complexity can only be achieved when dense vector or hash table is used, while this will result in high memory consumption and high CPU cache misses. Otherwise, when sparse data structure is used, it needs $O(logK_w)$ and $O(logK_d)$ complexity to get value from $N_{w|k}$ and $N_{k|d}$, respectively. Lastly, random access to $N_{w|k}$ and $N_{k|d}$ in MH step incurs high CPU cache misses, especially when K is large. It becomes worse since cyclic doc/word proposals are used and multiple MH steps are required. The analysis is validated by experiments in Section 7.

4 CGS Parallelization

Consider a typical web-scale application with millions of documents and words, there would be trillions of parameters with thousands of topics. No single machine can hold the entire Big corpus data nor the Big model size. This made single machine solution impossible, which motivates a scalable and efficient way of distributing the computation across multiple machines. However, the design is challenging that a typical web-scale LDA training requires both data parallelism and model parallelism and involves hundreds of machines.

4.1 Parallelization design

In this section, we will discuss multiple design dimensions to parallelize LDA training.

Synchronization approach. The first design choice we face is how to synchronize the model (across partitions) that not only eases the implementation on data-parallel platform but also gets similar converged accuracy. CGS algorithm itself is a serial process, since there are read-write dependence on N_k, $N_{w|k}$ and $N_{k|d}$. The dependence must be guarded by locks, which is costly and hard to implement in distributed environment. All existing systems have relaxed the locks on N_k consider that N_k is large and a single update is negligible. And the synchronization on $N_{w|k}$ and $N_{k|d}$ largely depends on the partitioning strategy, with two different approaches in literature: 1). Avoid dependence conflicts among different partitions by not partitioning common words and common documents into different partitions. For example, given p computing nodes, Peacock [31], NomadLDA [35] and SparkLDA [25] chose to partition the training data and model into $p*p$ partitions in a "diagonal" way, which are executed in p stages and in each stage p partitions are scheduled to be executed in parallel and the scheduler ensures there is no conflict among them. However, this largely increases the system overheads and entire model needs to be synchronized in each stage end thus with p times more network I/O. 2). Permit staleness on either $N_{k|d}$ or $N_{w|k}$. For example, parti-

	ZenLDA			ZenLDAHybrid			AliasLDA	
Decomposition	$\frac{\alpha*\beta}{N_k+W\beta}+\frac{N_{w\|k}*\alpha}{N_k+W\beta}+\frac{N_{k\|d}*(N_{w\|k}+\beta)}{N_k+W\beta}$			ZenLDA (hot words) or SparseLDA (tail words)			$\alpha(\frac{N_{w\|k}+\beta}{N_k+W\beta})+N_{k\|d}(\frac{N_{w\|k}+\beta}{N_k+W\beta})$	
Sampler	Alias	Alias	CDF	Alias	Alias	CDF	Alias	Alias
Fresh	no	no	no	no	no	no	no	yes
Computing	$O(1)^*$	$O(1)^*$	$O(K_d)$	$O(1)^*$	$O(1)^*$	$O(\min(K_d,K_w))$	$O(1)^*$	$O(K_d)$
Sampling	$O(1)$	$O(1)$	$O(\log K_d)$	$O(1)$	$O(1)$	$O(\min(\log K_d,\log K_w))$	$O(\#MH)$	$O(\#MH)$
Process Order	Word-by-Word			Hybrid			Doc-by-Doc	
Approximation	yes			yes			yes	

	LightLDA		F+LDA1		F+LDA2		SparseLDA		
Decomposition	$\frac{N_{w\|k}+\beta}{N_k+W\beta}*(N_{k\|d}+\alpha)$		$\alpha(\frac{N_{w\|k}+\beta}{N_k+W\beta})+N_{k\|d}(\frac{N_{w\|k}+\beta}{N_k+W\beta})$		$\beta(\frac{N_{k\|d}+\alpha}{N_k+W\beta})+N_{w\|k}(\frac{N_{k\|d}+\alpha}{N_k+W\beta})$		$\frac{\alpha*\beta}{N_k+W\beta}+\frac{N_{k\|d}*\beta}{N_k+W\beta}+N_{w\|k}(\frac{N_{k\|d}+\alpha}{N_k+W\beta})$		
Sampler	Alias	Alias/Lookup	F+Tree	BSearch	F+Tree	BSearch	LSearch	LSearch	LSearch
Fresh	no	no/yes	yes	yes	yes	yes	yes	yes	yes
Computing	$O(1)^*$	$O(1)$	$O(\log K)$	$O(K_d)$	$O(\log K)$	$O(K_w)$	$O(1)^*$	$O(1)^*$	$O(K_w)$
Sampling	$O(\#MH)$	$O(\#MH)$	$O(\log K)$	$O(\log K_d)$	$O(\log K)$	$O(\log K_w)$	$O(K)$	$O(K_d)$	$O(K_w)$
Process Order	Word-by-Word		Word-by-Word		Doc-by-Doc		Doc-by-Doc		
Approximation	yes		no		no		no		

Table 2: Comparison of different LDA sampling approaches. $O(1)^*$ means amortized complexity.

tion approach that all tokens corresponding to a word or a document are located in the same partition, thus only $N_{k|d}$ or $N_{w|k}$ is not synchronized in time, respectively, thus staleness is introduced. LightLDA [36] further blocks a partition into mini-batches in a "conjugated" way [3], and synchronization happens across mini-batches to reduce the state staleness.

As a comparison, we reduce the extra system complexity and remove the partition limitation. In this way, better performed partition strategy is allowed without introducing any extra system complexity. We achieve this by aggressively delaying the model update that all three model states are independently updated in parallel and are only synchronized at the epoch end. Furthermore even inside a partition, local model update is skipped. This `delayed update` essentially converts CGS algorithm to be a Monte-Carlo Expectation Maximization (MCEM) algorithm [10], that includes two steps:

- `E-step`. Each partition is executed independently, and topic is sampled with staled model state of last iteration. E-step does not update model state at all;
- `M-step`. Model state is updated with the aggregated value computed in E-step.

MCEM also completely avoids the lock cost if multi-threaded is enabled inside a partition. Therefore, no MH step is needed anymore that the accept probability always equals to 1. Besides, this introduced more opportunities for computing redundancy elimination. We have implemented MCEM for different CGS algorithms and the evaluation indicates that MCEM achieves similar converged accuracy, especially when corpus and the number of topics is large.

Graph based data and model representation. In stead of representing data (input corpus) and model (word-topic and document-topic) as (sparse) matrix, we represents data as a directed bipartite graph that is the dual representation of sparse matrix. Figure 1 depicts the graph representation of a corpus with three words (w_1, w_2, w_3) and documents (d_1, d_2, d_3). The graph has two kinds of vertices, word vertex and document vertex. An edge exists from word vertex to document vertex only if that word is occurred in the document. This representation is actually a natural graph [13] like many other natural language processing problems, where the graph have highly skewed power-law degree distributions.

Each word vertex is attached with the corresponding word-topic array ($N_{w|k}$) as attributes that word-topic matrix ($N_{w|k}$) is split in a word-wise fashion. Similarly, each document vertex is attached with the corresponding document-topic array ($N_{k|d}$). Current sampled topic (Z_{dw}) of a token is annotated as the corresponding edge attribute. Note that word-topic and document-topic array are sparse that the count for some topics is zero, and they become more and more sparse as the training converged. Relatively speaking, a long-tail word may have more sparse $N_{w|k}$ than a hot word; and $N_{k|d}$ maybe more sparse than $N_{w|k}$ since a word may have more occurrences than the average document length. It is noteworthy that there may be multiple occurrences of the same word in one document. Instead of representing them as one edge annotated with composed topic array (array[int]), we represent them separately that each token has one edge annotated with the sampled topic (int). This largely reduces two-thirds of edgeRDD size since the extra array meta data is removed (the size of array with one int-typed element is 12 bytes). And the global state $N_k = \sum_d N_{k|d} = \sum_w N_{w|k}$ is a global variable whose value is computed by aggregating the $N_{w|k}$ from all word vertices (we do not aggregate it from $N_{k|d}$ since W is typically 100X less than D).

Partition approach. Parallelism is achieved by partitioning the graph into multiple partitions, and workers apply local CGS process for each individual partition in parallel,

[3] If a partition is document-wise partitioned, then the mini-batch is word-wise, or vice versa.

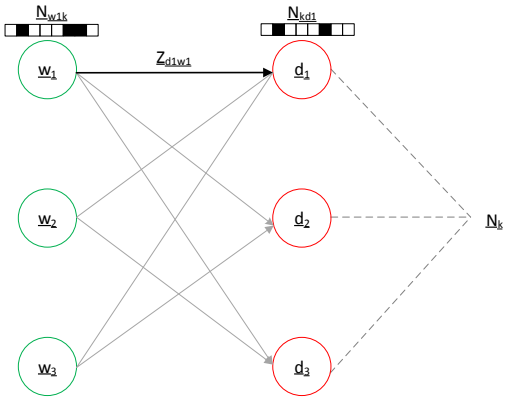


Figure 1: Graph based CGS abstraction.

followed by synchronizing the model state at the iteration end. In this way, both data parallelism and model parallelism are achieved [31], where the model is also partitioned and distributed across workers.

How to partition the corpus and model plays crucial impact on system performance as well as model staleness (described in last section). The improper partition would result in load imbalance and large network communication. Compared with (sparse-)matrix based representation that can only be partitioned in a "rectangle" way, graph has more freedom for partitioning choice. GraphX supports vertex-cut [13] that each vertex has a master and network I/O is introduced to synchronize the vertex attribute (model state) between vertex master and slaves. Note that in GraphX, edges in a partition have been grouped together according to the source vertex (word). Currently GraphX provides several common used partitioning approaches, however, they do not fit well for power-law graph partition. Instead, we present an improved degree-based hash partition (DBH+) algorithm that extend DBH [32], and the algorithm is listed in Algorithm 3. DBH is proved to achieve lower communication cost than existing methods and simultaneously guarantee good workload balance dealing with power-law graph. It first applies randomized hash function to evenly assign vertices to partitions, then assigns an edge to a partition that contains its source or destination vertex who has smaller degree. DBH shares the same insights with PowerLyra [11] that locality matters for low-degree vertex thus it places all edges related to this vertex together, while parallelism matters for high-degree vertex thus it favors to cut high-degree vertex. However, DBH only considers the relative size between source degree and destination degree, without considering their absolute value. Consider the case where both source and destination degree are small (smaller than a threshold value), it is not reasonable to still correspond the edge to vertex with lower degree, but should be the vertex with higher degree. Base on our experience, document-wise partition can usually yield similar performance with DBH since word degree (especially for hot words) is usually larger than document degree thus DBH will also make the same partition choice. DBH works better for tail words that has lower degree than document, while DBH+ lies in between by introducing a degree threshold that provides more flexibility for performance tuning.

Algorithm 3 DBH+: improved degree-based hash partition.

Input: Edge set E; vertex ser V; machines set P.
Output: Assignment $P(e) \in [P]$ for each edge e.
procedure DBHPLUS
  for each $v \in V$ **do**
    $P(v) = hash(v)$
  count the degree d_i for each $i \in V$ in parallel
  for each $e = (v_i, v_j) \in E$ **do**
    if $max(d_i, d_j) < threshold$ **then**
      if $d_i \leq d_j$ **then**
        $P(e) = P(d_j)$
      else
        $P(e) = P(d_i)$
    else
      if $d_i \leq d_j$ **then**
        $P(e) = P(d_j)$
      else
        $P(e) = P(d_i)$

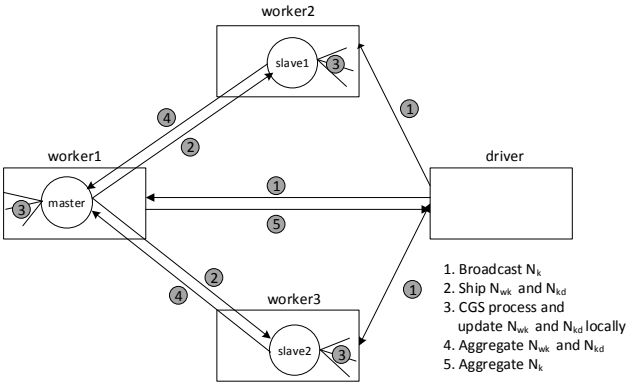


Figure 2: ZenLDA workflow in an iteration.

Training workflow. The workflow of one iteration is illustrated in Figure 2. It can be logically split into five steps: 1) driver broadcasts N_k to all workers. 2) the vertex master ships the model state ($N_{k|d}$ or $N_{w|k}$) to all of the corresponding vertex slaves. 3) workers apply local CGS step in parallel, where word-by-word process order is chosen in a partition for better cache locality that $N_{w|k}$ is reused and $N_{k|d}$ has spatial locality among different topics. $N_{k|d}$ and $N_{w|k}$ is locally updated at the end. 4) at the end of an iteration, vertex master aggregates $N_{k|d}$ or $N_{w|k}$ all local updates from slaves. 5) driver aggregates $N_k = \sum_w N_{w|k}$ from all word master vertices. E-step includes step 2 and 3, and it is implemented by *VetexRDD.aggregate*, and M-step is performed in step 4 via *graph.aggregateMessages*. Almost all steps except step 3 have network communication. The size of N_k in step 1 is negligible, so we pay more efforts on reducing the size of $N_{k|d}$ and $N_{w|k}$ (Section 5).

4.2 Related work in CGS parallelization

Another trend in literature [25, 36, 35, 24, 5, 30] is to parallelize the CGS in distributed environment. Besides the different synchronization choice we make, how to implement the synchronization (i.e., the system choice) also differs. Almost of existing distributed LDA systems resort to implement the synchronization via MPI except LightLDA that relies on a dedicated parameter server. They will increase the system complexity and engineering effort, consider that they need manually to do data partitioning, pipeline execution, fault tolerance via checkpoint, and task scheduling, etc. As a comparison, these complexities in our work are hidden by Spark framework that we only focus on the core CGS logic (E-step) while the remaining steps can be simply done by the corresponding APIs. Almost all works [25, 36, 35] partition the corpus in a document-wise way, while we permit any kind of partition methods.

5 Optimizations

In this section, we present several optimizations to push performance to the limit .

5.1 Approximated training

Sparse model initialization. The first several iterations are always the performance and scalability bottleneck, since model ($N_{k|d}$ and $N_{w|k}$) becomes more sparse as training makes progress. Usually, model is initialized by first randomly sampling a topic for each token with equal probability, followed by aggregating the topic distribution to initiate the model state. However, such `random initialization` would result in relatively dense topic distribution for word especially hot words that occurred in most of the documents. This dense word-topic distribution takes more storage, memory, network I/O (step 2 in Figure 2) and more computing complexity as well. Instead of random initialization, we presents two `sparse initialization` approaches that demonstrate better performance in the first several iterations and achieves comparable or even better accuracy (See Figure 11): 1). `Sparsify word-topic array` ($N_{w|k}$) directly with sparsity degree $deg \ll 1$. Given T tokens of word w in the corpus and topic number K, it first randomly samples topic set S from K with $deg * K$ topics, then randomly samples a topic from $k \in S$ for each token of that word with equal probability, and updates $N_{k|d}$ and $N_{w|k}$ accordingly. This would introduce side effects on model accuracy. On the one hand, it reduces the possibility to allocate the same topic for two words that should be with different topic, since their topic overlapping probability is reduced due to sparse initialization. On the other hand, this also reduces the possibility if they should be with the same topic. In other words, it could have better word log-likelihood consider that the probability of any two words with the same topic is small. This optimization is essentially to gradually amortize the cost of first iteration to the following iterations. The side effect can be neutralized by increasing the β value in decomposed part $N_{k|d} * (N_{w|k} + \beta)$ for those topics that are not assigned during initialization. 2). `Sparsify document-topic array` ($N_{k|d}$) similarly that indirectly results in sparse word-topic array.

"Converged" token exclusion. We observed that different tokens are with different convergence rate, it is unnecessary to sample for these converged token repeatedly. We present `"Converged" token exclusion` that excludes converged tokens thus largely reduces the workload per iteration, especially for later iterations when most of the tokens are almost converged (See Figure 11a). A token is considered to be converged if current sampled topic is the same as topic sampled in last iteration. To reduce the side effect, we do not simply exclude the converged token, but exclude them with a probability. We include them into sampling based on how many iterations a token has not been processed (i) and how many times it was processed but with the same sampled topic (j). It is computed by 2^{i-j} with positive correlation with i but with negative correlation with j. i will be zeroed if the token is sampled, otherwise will increase by one in each iteration; j is cleared once the sampled topic is changed, otherwise will increase by one with probability $1/2$. This optimization is not used at the beginning, but enabled after certain iterations when the model is largely converged.

5.2 I/O reduction via delta aggregation

We further reduce network I/O in step 4 where each vertex slave sends its locally aggregated $N_{k|d}$ and $N_{w|k}$ to master. With the insight that high proportion of tokens are converged without topic change, we present `delta aggregation` that only changed tokens are locally aggregated thus network I/O is largely reduced as the model becomes converged. This requires to keep the old topic sampled last time and new sampled topic, thus doubles the edge attribute size. And the effectiveness would be offset by "converged" token exclusion. Thereby, we disable this optimization if token exclusion is enabled. Note that unlike token exclusion, it does not affect model accuracy.

5.3 Low-level optimizations

Besides system design, we also present several low-level optimizations, including data structure that exploits sparsity and redundant computing elimination. [4]

Sparse data structure. Inherent sparsity exists in word-topic array $N_{w|k}$ and document-topic array $N_{k|d}$, while different sparsity degree would require different data structure. Besides `DenseVector` and `SparseVector` pro-

[4] We also implement optimizations described in LightLDA to exploit the difference between hot and long-tail word.

vided in MLlib, we present `CompactVector` that fits data with medium sparsity. Take vector $(1,0,0,0,0,3,0)$ with 7 elements as an example. It is represented in dense format as $[1,0,0,0,0,3,0]$, while represented in sparse format as $(7,[0,5],[1,3])$, where 7 is the size of the vector, $[0,5]$ is the index array that records the indices of non-empty elements, and $[1,3]$ is value array that records the corresponding values. SparseVector is more memory efficient if vector has large sparsity, but its cost for search operation is increased from $O(1)$ to $O(\log(length))$, and it would result in bigger size if sparsity degree is small. The tipping point is at sparsity with 0.5 where half of elements are empty, thus the total length of index and value array is the same as original length. Compared with SparseVector, our `CompactVector` has the same value array but with a different index array that is composed of (s,n) pairs where s records the starting index of an empty sequence and n records the number of non-empty elements before position s. With regard to this example, it is represented as $(7,[(1,1)],[6,1])$. Consider a vector with M non-empty sequences and N non-empty elements ($M \leq N$). The size of CompactVector format could be smaller than SparseVector when $\frac{N}{M} \geq 2$. Getting value from CompactVector has $O(\log M)$ complexity, which is lower than $O(\log N)$ in SparseVector, while insertion in CompactVector is more costly with $O(N)$ complexity. The right choice should tradeoff between space requirement and computing cost. Generally, SparseVector is suitable for vectors with large sparsity; DenseVector is suitable for dense vector with many write operations; and CompactVector is suitable for situations with medium sparsity and almost operations are read.

Alias table. The time complexity to build alias table for $\frac{\alpha*\beta}{N_k+W\beta}$ ($gTable$) and $\frac{N_{w|k}*\alpha}{N_k+W\beta}$ ($wTable$) is $O(K)$ and $O(K_w)$, respectively. The cost of $gTable$ is amortized since it is built once and used afterwards in one iteration. And we reduce the memory consumption of $wTable$ by processing the tokens in word-by-word fashion that unused $wTable$ would be freed (GC). To reduce the creation cost, we further refine the algorithm presented in AliasLDA [20]. First, we only maintain the `high` (H) queue that keeps the topic information ((k,p_k)) that is with higher probability than the average $\frac{1}{K}$, and do not maintain the opposite `low` (L) queue. Instead, we directly insert the topic information in L queue into the corresponding bin in alias table. Second, in some cases we can keep probability as integer rather than float to save memory cost. For instance, when creating alias table for $N_{k|d}$ (used in LightLDA), the probability (count) is integer, but the average probability would be float (dividing by K_d). We actually convert it to integer by multiplying K_d for each individual topic probability in $N_{k|d}$.

Redundant computing elimination. Redundant computing exists in E-step. For instance, $\frac{1}{N_k+W\beta}$ will be used many times during entire iteration, thus we can precompute it first and re-use the result later. This also benefit CPU cache usage with reduced memory footprint. It is noteworthy that the delayed model update in MCEM (model state never change in E-step) exposes more redundant computing. Besides, ZenLDA tries to reuse the same generated random number to avoid cost of random number generation, consider that there are three random number generations per token sampling.

6 Implementation

The implementation can be found at `https://github.com/cloudml/zen/`. We have resolved the encountered inefficiency of managed language (Scala) and framework cost of GraphX. The implementation balances resource (CPU, network and memory) usage such that no resource is the bottleneck and all are fully utilized as far as possible.

Memory. Memory is the major bottleneck when we first try to scale out LDA training. Data-parallel system like Spark is designed to process one partition per core and the whole partition must be loaded in memory. "Out of memory" occurs frequently if many partitions (16-32 cores per machine) are loaded at the same time while CPU is under-utilized. Simply reduce the partition size does not work well, since too many partitions would incur increased network I/O among partitions. We observed that partitions in a machine may share common data, such as the same word or document and the corresponding model state may exist in multiple partitions. Therefore, we choose to increase the partition as large as possible, load less partitions (less than cores) at one time and use multi-threading in a partition to fully utilize the CPU cores. More specifically, edges are sorted and queued in a partition (already done by GraphX), and they are processed in word-by-word order. Once a thread completed one word, it fetches all edges of the first word in queue (work-stealing) for processing. This achieves relative good load balance. We also re-implement some GraphX APIs (except shuffling operator) to make them multi-threaded, such as `ShipVertexAttributes` (step 2) and `aggregateMessages` (step 4), otherwise the serial part would be the new bottleneck. Besides, GraphX APIs would create many intermediate objects that raise higher memory consumption and GC overhead. Thus we choose to directly operate on *Graph* data structures and even modify them (e.g., edge array and vertex index array in EdgePartition, vertex array and routing table in VertexPartition). For instance, original EdgePartition stores the source vertex ID (word ID) for each token that has large redundancy, since the number of tokens is much larger than words number. Consider that edges are al-

Dateset	Tokens	Words	Docs	T/D
NYTimes	99,542,125	101,636	299,752	332
PubMed	737,869,083	141,043	8,200,000	90
BingWebC1Mon	3,150,765,984	302,098	16,422,424	192
BingWebC320G	54,059,670,863	4,780,428	406,038,204	133

Table 3: Four different datasets used in evaluation.

ready sorted word-by-word, we just store word ID and corresponding starting offset for each word, which saves about 1/3 EdgePartition size. Besides, we actually abandon `aggregateMessages` that is used to update vertex attribute with value aggregated from edges, since it constraints the edge attribute type must be the same as vertex thus introduces costly type conversion.

CPU. Once we fixed the memory limitation via multi-threading, we found that high CPU cost on RDD decompression and deserialization that is processed by a single thread. Therefore, we prefer to configure RDD in uncompressed and deserialized format. In addition, carefully programming in Scala to avoid boxing/unboxing and generation of closures can also reduce the CPU cost. For example, to represent the bin of alias table $((i,h,p_l))$, we use three arrays with primitive type instead of one array with Scala `Tuple3` to avoid the boxing/unboxing overhead.

Network. The model shuffling cost is a critical factor of performance, especially when model size and partition number are large. Besides the sparse representation, we compress model using JavaFastPFOR [19], a compression library specially useful for integer, to reduce storage and shuffle cost. It achieves much higher compression rate and runs much faster than generic compressors in Spark. We further encapsulate SparseVector as `CompressedVector`. We only decompress it during sampling, otherwise we keep it in compressed form. Besides, we adopt Kryo serialization library in Spark that is significantly faster and more compact than default Java serialization.

7 Evaluation

7.1 Evaluation design

Datasets. We use four different datasets, including a small sized NYTimes [4] (520MB), a medium sized PubMed [3] (3.8 GB), a large one month web chunk data indexed by Bing News (BingWebC1Mon, 17GB), and a super large-scale Bing data (BingWebC320G). They are all pre-processed and saved as `libsvm` format. The detailed information is listed in Table 3.

Evaluation design. The evaluation aims to evaluate: 1) the effectiveness and efficiency of ZenLDA compared with other CGS algorithms. 2) the effectiveness and efficiency of MCEM algorithms compared with the corresponding CGS version. 3) the scalability of ZenLDA that varies topic number, dataset size and number of machines. 4) the effectiveness of our proposed techniques.

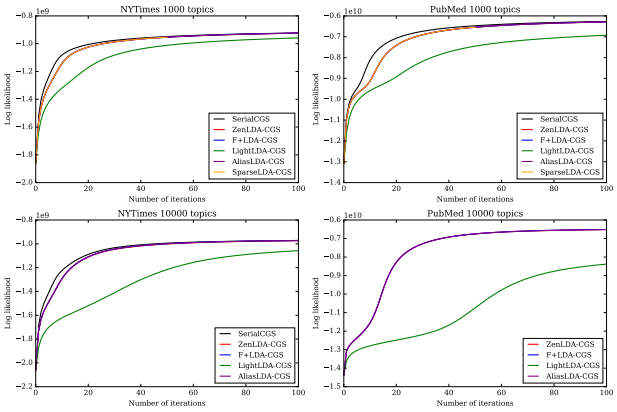


Figure 3: Log-likelihood comparison among different CGS algorithms.

	NYTimes K=1,000	PubMed K=10,000	BingWebC1Mon K=100,000
ZenLDA	1162.3	1654.7	5732.1
DMTK	1424.0	1694.0	7314.0

Table 4: Comparison between ZenLDA and DMTK: total training time (in seconds) in 100 iterations (NYTimes/PubMed) or 60 iterations (BingWebC1Mon).

Cluster configuration. We have three Spark clusters at different scale. The smallest one has 8 homogeneous computing nodes that are connected via 1Gbps Ethernet. Each node has 12 2.10GHz Intel(R) Xeon(R) E5-2620 cores with multi-threading enabled. Driver is configured with 4GB memory and 8 workers are configured with 40GB memory. The experiments against NYTimes, PubMed are conducted in this cluster, where NYTimes is executed in a single machine with 3 partitions that each partition has 8 threads, while PubMed is executed using all 8 machines, split into 24 partitions that each partition has 8 threads. The medium cluster has 10 homogeneous computing nodes that are connected via 40Gbps Infiniband network and each node has 16 2.40GHz Intel(R) Xeon(R) CPU E5-2665 cores. The driver is configured with 5GB memory and 10 workers are configured with 100GB memory. BingWebC1Mon is evaluated in this medium cluster with 40 partitions that each partition has 4 threads. The largest Spark cluster is deployed on a multi-tenancy data center managed by Yarn [29] that the resource is not always guaranteed, where an executor is configured to have 20GB memory and 10 cores. The scalability experiments against BingWebC320G are conducted in this cluster.

7.2 Evaluation on ZenLDA algorithm and MCEM parallelization

ZenLDA evaluation. We first compare different CGS algorithms. We have implemented SparseLDA, AliasLDA, LightLDA, F+LDA (See Table 2) and ZenLDA using the same framework on Spark, with only 174, 203, 209, 191

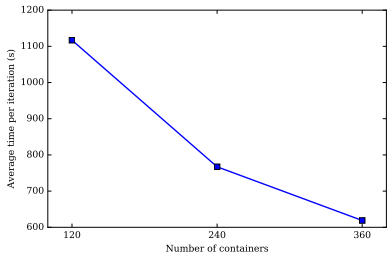


Figure 7: Execution time change curve as executor number varies.

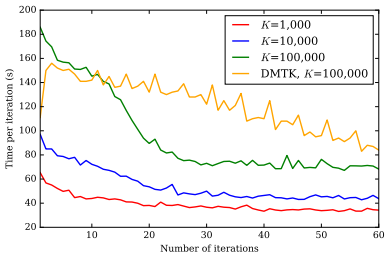


Figure 8: Execution time change curve as topic number varies.

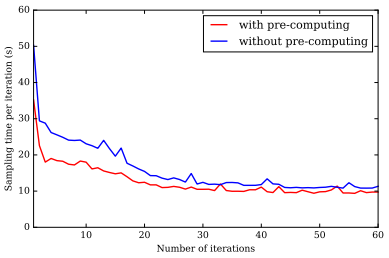
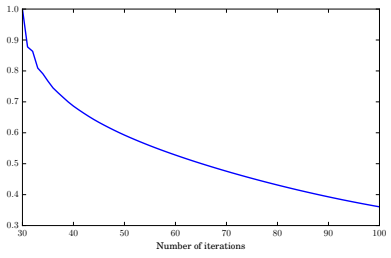
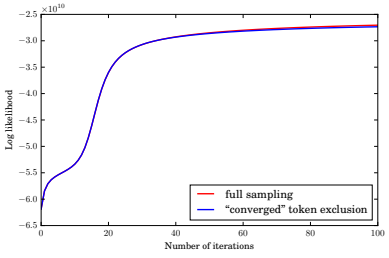


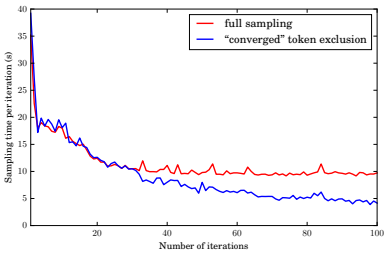
Figure 9: Execution time evaluation on redundant computing elimination.



(a)



(b)



(c)

Figure 11: (a) Sample rate of token's topic assignments. (b) Sampling time evaluation on "converged" token exclusion. (c) Corresponding log-likelihood evaluation.

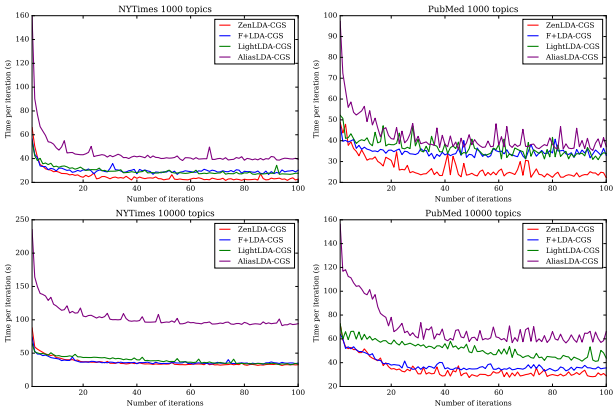


Figure 4: Comparison among different CGS algorithms: time-iteration curve.

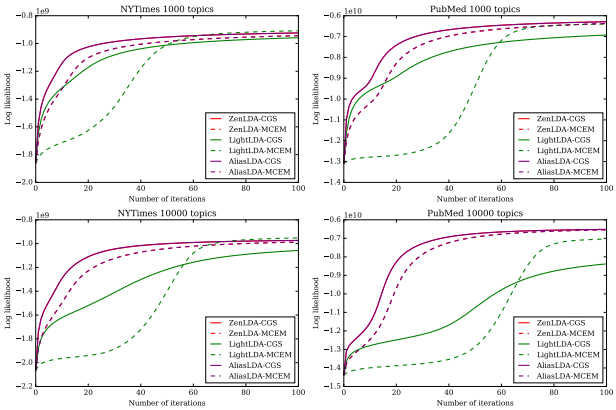


Figure 5: Log-likelihood comparison between CGS and MCEM.

and 337 lines of Scala code, respectively. All implementations are with asymmetric prior and are applied the same optimizations (Section 5 and Section 6), except sparse initialization and "converged" token exclusion that will be evaluated in next section. Each proposal distribution in AliasLDA and LightLDA is with only one MH step [5]. We cannot compare SparkLDA [25] since it is not open-sourced, but we believe ZenLDA will win since SparkLDA uses standard CGS algorithm. We did not report the comparison with EM based LDA implementation in MLlib, which even cannot finish the first iteration against PubMed dataset with errors. The comparison is against NYTimes and PubMed datasets with 1,000 and 10,000 topics, respectively. All experiments have the same α and β as 0.01. Each experiment is executed with 100 iterations. Both execution time per iteration and log-likelihood per iteration are compared. Figure 3 illustrates the comparison on model convergence. `SerialCGS` represents the accuracy baseline that is evaluated by open-sourced serial F+LDA implementation [2]. And among different CGS algorithms, `SparseLDA` and `F+LDA` are almost the same as baseline since no local approximation is applied. AliasLDA also shares the similar accuracy (hard to distinguish) since the proposal distribution is almost the same as original distribution. As a comparison, ZenLDA converges slowly at the beginning (hard to distinguish), but quickly catches up later. ZenLDA actually converges faster than them if likelihood-walltime curve

[5] They did not report the best number of MH steps. More MH steps, more performance slowdown with possible accuracy improvement.

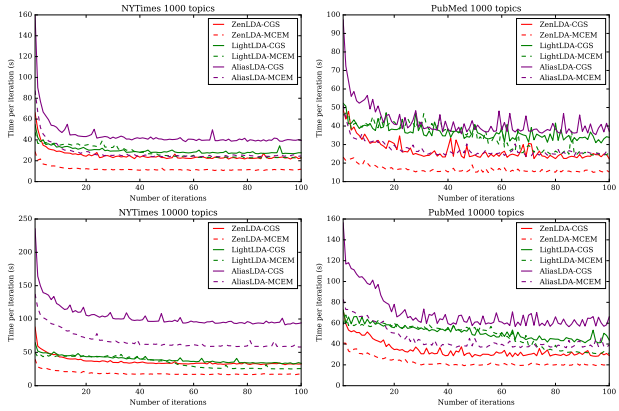


Figure 6: Time cost comparison between CGS and MCEM.

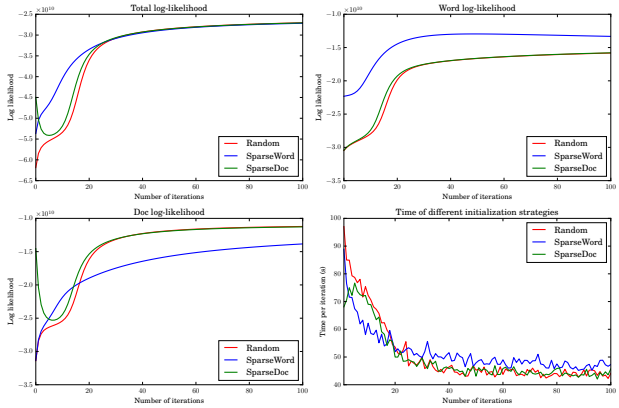


Figure 10: Comparison among different model initialization.

is drawn (combine Figure 4 and Figure 3). LightLDA is the worst with respect to log-likelihood, and even much worse as the number of topics increased. This may be due to the proposal distribution used in LightLDA is far from the true probability. Figure 4 depicts the execution time (y-axis) of each iterations (x-axis). We exclude the log-likelihood computing time, and the spikes in Figure 4 stems 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 204.8s on smallest data (NYTimes) with smallest topics ($K = 1000$). The evaluation validates the comparison with $SparseLDA < AliasLDA < F+LDA < ZenLDA$ that ZenLDA is the best performer and the speedup keeps almost the same among different dataset and different topics number. The performance of LightLDA is a little bit "surprising", consider that it has only $O(1)$ complexity and with only one MH step. With more data and more topics, LightLDA even performs worse. The major factor is due to the slow convergence of LightLDA with more dense model thus larger network I/O. Besides, there are two implementation related reasons: 1) we implement alias table rather than look-up table for document proposal since data is not partitioned in document-wise; 2) as we discussed in Section 3.2, the MH-step in LightLDA would be costly due to the computation of accept probability, which requires $O(\max(\log K_w, \log K_d))$ complexity since the sparse representation of $N_{k|d}$ and $N_{w|k}$ has $O(\log K_w)$ and $O(\log K_d)$ complexity to read the value, respectively. We do not represent them as hash table since it requires more memory space and results in more cache misses.

Evaluation on MCEM parallelization. For each algorithm we have implemented two parallelized versions, MCEM with delayed update and CGS with in time local update. SparseLDA is ignored because it is relatively slow and F+LDA is also skipped since F+ tree is not designed for delayed update (MCEM) but for instant update since it has low update cost. The model convergence comparisons are shown in Figure 5. Note that the curves of ZenLDA and ZenLDA-MCEM are almost covered by that of AliasLDA and AliasLDA-MCEM. The evaluation indicates that MCEM with delayed update still converges with similar accuracy. Compared with CGS implementation with local update, the MCEM variant converges a little bit slower at first but finally catches up after 60˜80 iterations (AliasLDA-MCEM vs. AliasLDA, ZenLDA-MCEM vs. ZenLDA). LightLDA behaves differently that its MCEM version converges much slower at first but catches up and even outperforms after the 63th iteration. Figure 6 depicts the execution time comparisons. We can see that MCEM significantly outperforms the corresponding CGS version, since the locks in local update are avoided, MH-steps are skipped in AliasLDA and ZenLDA, and some pre-computing can be applied. It also indicates that in all experiments execution time decreases as model becomes more sparse until converged, and the first several iterations are always the most time-consuming parts.

Comparison with DMTK We have also compared MCEM version of ZenLDA (the best performer in Spark) with DMTK on the medium cluster. DMTK is considered as the state-of-art LDA training system that implements LightLDA ($O(1)$ complexity) algorithm on top of parameter server with 2,925 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 is against on all datasets excepts BingWebC320G.

With respect to performance, Table 4 shows that ZenLDA achieves similar performance as DMTK in NYTimes and PubMed datasets, and even 27.6% faster (95.5s vs. 121.9s) in larger BingWebC1Mon dataset with 100,000 topics. This is because the ratio of fixed cost in ZenLDA is largely reduced in large dataset. More specifically, ZenLDA has much better performance in the first several iterations, and DMTK gradually catches up afterwards as the model becomes more sparse thus the communication cost is reduced. The insight can be shown in Fig-

ure 3 that LightLDA converges much worse than ZenLDA at first, and the worse convergence would also result in larger communication cost thus poor performance. This supports our claim that model accuracy and system performance are correlated each other and they must be carefully balanced. As a comparison, in ZenLDA only step 2 (Figure 2) has gradually reduced shuffling cost but has almost fixed cost in other steps. The performance difference between two LightLDA implementations (Spark 4 vs. DMTK) indicates the language cost (C++ vs. Scala) and framework cost (Parameter Server vs. Spark). We believe that ZenLDA can get more speedup if it is implemented in C++ but will introduce more engineering cost.

7.3 Scalability

The scalability experiments are conducted only for ZenLDA against BingWebC320G dataset with 10,000 topics and run on Bing multi-tenancy data center. Each partition is executed in an executor (container) with 10 threads. Figure 7 indicates that ZenLDA scales pretty well. With 2X more executors (240 versus 120 containers), the performance is almost linearly improved. As we continue to add more executors (360), the performance can still be improved, but with less speedup as network I/O becomes larger. Note that during the experiments some failures occur. For instances, some machines are taken away randomly thus ZenLDA needs to retry from last checkpoint. The fault or straggler tolerance supports in Spark makes ZenLDA more promising in production that needs large scalability, easy deployment and execution in shared environment.

We also evaluated the performance when topic number varies. The experiment is conducted against BingWebC1Mon with 1K, 10K and 100K topics, respectively. Larger topic number has larger shuffling cost that is gradually reduced until model is converged. Their training time of first 60 iterations is shown in Figure 8. The stable average time per iteration is only increased from 34s to 44s when topic number increases from 1K to 10K. Even with 100X more topics (100K), the time per iteration is only increased to 69s. So ZenLDA is very scalable when topic number increases.

7.4 Optimization evaluation

This section evaluates the effectiveness of optimizations presented in Section 5 that is applied on the MCEM parallelization of ZenLDA.

Sparse initialization. We set the sparseness of $N_{w|k}$ and $N_{k|d}$ as 0.1. Figure 10 shows the log-likelihood of different initialization strategies (a), as well as the specific word log-likelihood (b) and document log-likelihood (c) (We use the same log-likelihood decomposition as in [27]). They both achieves almost the same converged accuracy as random initialization. It is within expectation that sparse initialization of word-topic distribution (`SparseWord`) achieves much better word log-likelihood, but with worse document log-likelihood. In contrast, sparsifying document-topic distribution (`SparseDoc`) has better document log-likelihood. With respect to performance, Figure 10 (d) shows that both SparseWord and SparseDoc make the sampling faster than random initialization at the first several iterations. This is helpful to reduce the scalability bottleneck. However, it gradually increases to normal performance as random initialization as we expected, and even higher in SparseWord because of the increased K_d (the worse document log-likelihood, the dense document-topic distribution).

"Converged" token exclusion. ZenLDA turns on this optimization after the 30th iteration. Both sampling time (step 3) and log-likelihood are compared. The result shown in Figure 11c and Figure 11b indicates that it can achieve about 2X speedup in later iterations and barely hurting the model accuracy. Figure 11a explains the underlying reason that the changing rate of topic assignment decreases as the iteration increases, with only about 37% of tokens have changed the sampled topic at the end. This figures also validates that delta aggregation (Section 5.2) can largely reduce the network I/O. The speedup is not strictly align with the change rate since sample rate also considers other factors.

Redundant computing elimination. We only evaluate "redundant computing elimination" and exclude optimization that avoids random number generation. The result in Figure 9 shows that the sampling is faster with up to 40% improvements at the beginning. The speedup decreases gradually since the time spending on CGS sampling also decreases. Besides avoiding the expensive locks, this is actually one of the reasons that MCEM variant performs better than CGS algorithm.

8 Conclusion

In this paper, we present techniques to provide an efficient and scalable CGS system on distributed data-parallel platform, and the proposed techniques are general useful in other system implementations. With combined algorithm innovation and system improvements, we demonstrate that build distributed machine learning system should combines indispensable innovations from both algorithm side and system side, and the distributed data-parallel abstraction (especially graph abstraction) is not only feasible and beneficial, but also efficient and scalable. We will continue this methodology and add more and more models in the future.

References

[1]

[2] NomadLDA. http://bigdata.ices.utexas.edu/software/nomad/.

[3] PubMed. http://www.ncbi.nlm.nih.gov/pubmed.

[4] Linked open data. http://data.nytimes.com/, 2015.

[5] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola. Scalable inference in latent variable models. WSDM, pages 123–132, 2012.

[6] Apache. What is Apache Mahout? http://mahout.apache.org/.

[7] O. ARB. Openmp specifications. http://openmp.org/wp/openmp-specifications/, 2013.

[8] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent Dirichlet Allocation. *JMLR*, 3:993–1022, 2003.

[9] J. Bradley. Topic modeling with LDA: MLlib meets GraphX. https://databricks.com/blog/2015/03/25/topic-modeling-with-lda-mllib-meets-graphx.html, 2015.

[10] J. Chen, K. Li, J. Zhu, and W. Chen. WarpLDA: a simple and efficient O(1) algorithm for Latent Dirichlet Allocation. arXiv:1510.08628, 2015.

[11] R. Chen, J. Shi, Y. Chen, and H. Chen. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. EuroSys, pages 1:1–1:15, 2015.

[12] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. OSDI, pages 10–10, 2004.

[13] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI 12*, pages 17–30, 2012.

[14] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613, 2014.

[15] T. L. Griffiths and M. Steyvers. Finding scientific topics. *PNAS*, 101:5228–5235, April 2004.

[16] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *NIPS 26*, 2013.

[17] T. Hofmann. Probabilistic latent semantic analysis. In *UAI*, pages 289–296, 1999.

[18] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. EuroSys, pages 59–72, 2007.

[19] D. Lemire. JavaFastPFOR: A simple integer compression library in Java. https://github.com/lemire/JavaFastPFOR.

[20] A. Q. Li, A. Ahmed, S. Ravi, and A. J. Smola. Reducing the sampling complexity of topic models. KDD, pages 891–900, 2014.

[21] 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. OSDI, pages 583–598, 2014.

[22] Z. Liu, Y. Zhang, E. Y. Chang, and M. Sun. Plda+: Parallel latent dirichlet allocation with data placement and pipeline processing. *TIST*, 2(3):26, 2011.

[23] X. Meng, J. K. Bradley, B. Yavuz, E. R. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. MLlib: Machine Learning in Apache Spark. *CoRR*, 2015.

[24] D. Newman, A. Asuncion, P. Smyth, and M. Welling. Distributed algorithms for topic models. *JMLR*, pages 1801–1828, 2009.

[25] Z. Qiu, B. Wu, B. Wang, and L. Yu. Gibbs collapsed sampling for latent dirichlet allocation on spark. volume 36, pages 17–28, 2014.

[26] M. Research. Distributed machine learning toolkit. https://github.com/Microsoft/DMTK.

[27] A. Smola and S. Narayanamurthy. An architecture for parallel topic models. *Proceedings of the VLDB Endowment*, 3(1-2):703–710, 2010.

[28] S. Tora and K. Eguchi. MPI/OpenMP hybrid parallel inference for latent dirichlet allocation. LDMTA, pages 5:1–5:7, 2011.

[29] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache

Hadoop YARN: Yet another resource negotiator. SOCC, pages 5:1–5:16, 2013.

[30] Y. Wang, H. Bai, M. Stanton, W.-Y. Chen, and E. Y. Chang. PLDA: Parallel latent dirichlet allocation for large-scale applications. AAIM, pages 301–314, 2009.

[31] 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 Trans. Intell. Syst. Technol.*, 6:47:1–47:23, July 2015.

[32] C. Xie, L. Yan, W. jun Li, and Z. Zhang. Distributed power-law graph computing: Theoretical and empirical analysis. In *NIPS 27*, pages 1673–1681.

[33] 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*, pages 1335–1344, 2015.

[34] L. Yao, D. Mimno, and A. McCallum. Efficient methods for topic model inference on streaming document collections. KDD, pages 937–946, 2009.

[35] H.-F. Yu, C.-J. Hsieh, H. Yun, S. Vishwanathan, and I. S. Dhillon. A scalable asynchronous distributed algorithm for topic modeling. WWW, pages 1340–1350, 2015.

[36] J. Yuan, F. Gao, Q. Ho, W. Dai, J. Wei, X. Zheng, E. P. Xing, T. Liu, and W. Ma. LightLDA: Big topic models on modest computer clusters. In *WWW*, pages 1351–1361, 2015.

[37] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.