WarpLDA: a Cache Efficient O(1) Algorithm for Latent Dirichlet Allocation

Jianfei Chen[†‡], Kaiwei Li[†§], Jun Zhu[†‡], Wenguang Chen[†§]
[†]Dept. of Comp. Sci. & Tech.; TNList Lab; CBICR Center; Tsinghua University
[‡]State Key Lab of Intelligent Technology & Systems, Beijing, 100084 China
[§]Parallel Architecture & Compiler Tech. of Mobile, Accelerated, and Networked Systems (PACMAN)
{chenjian14, likw14}@mails.tsinghua.edu.cn;{dcszj, cwg}@tsinghua.edu.cn

ABSTRACT

Developing efficient and scalable algorithms for Latent Dirichlet Allocation (LDA) is of wide interest for many applications. Previous work has developed an $O(1)$ Metropolis-Hastings (MH) sampling method for each token. However, its performance is far from being optimal due to frequent *cache misses* caused by random accesses to the parameter matrices.

In this paper, we first carefully analyze the memory access behavior of existing algorithms for LDA by cache locality at document level. We then develop WarpLDA, which achieves $O(1)$ time complexity per-token and fits the randomly accessed memory per-document in the L3 cache. Our empirical results in a wide range of testing conditions demonstrate that WarpLDA is consistently 5-15x faster than the state-of-the-art MH-based LightLDA, and is faster than the state-of-the-art sparsity aware F+LDA in most settings. Our WarpLDA learns a million topics from 639 millions of documents in only five hours at an unprecedented throughput of 11 billion tokens per second.

1. INTRODUCTION

Topic modeling provides a suite of statistical tools to discover latent semantic structures from complex corpora, with latent Dirichlet allocation (LDA) [7] as the most popular one. LDA has found many applications in text analysis [8, 36], data visualization [18, 22], recommendation systems [14], information retrieval [29] and network analysis [11, 13]. LDA represents each document as an admixture of topics, each of which is a unigram distribution of words. Since exact inference is intractable, both variational Bayes (VB) and Markov Chain Monte Carlo (MCMC) methods have been developed for approximate inference, including mean-field variational Bayes [7], collapsed variational Bayes [26], collapsed Gibbs sampling (CGS) [16] and expectation propagation [23]. Among these methods, CGS is most popular due to its simplicity and availability for fast sampling algorithms [30, 20, 32].

Entering the Big Data era, applications often require *large-scale* topic modeling to boost their performance. For example, Wang et al. [28, 32] show that learning 1 million topics can lead to significant performance gain on various tasks such as advertisement and recommendation. Other recent endeavours on learning large topic models often contain billions of documents, millions of topics, and millions of unique tokens [28, 32, 21]. Various fast sampling algorithms have been proposed for LDA, reducing the time complexity of sampling the topic assignment per-token from $O(K)$ to $O(1)$, where K is the number of topics [30, 21, 32].

Table 1: Memory hierarchy in an Intel Ivy Bridge CPU. L1D denotes L1 data cache, and * stands for per core.

	L1D	L2	L3	Main memory
Latency (cycles)	5	12	30	180+
Size	32KB*	256KB*	30MB	10GB+

Although many efforts have been spent on improving the per-token sampling complexity, little attention has been paid to examine the cache locality, another important dimension to improve the overall efficiency. For all the aforementioned algorithms, the cache locality is not getting better; in fact some are even getting *worse* (See Table 2 for details). The running time of LDA is often dominated by *random memory accesses*, where the time consumption is roughly proportional to the latency of each access. As shown in Table 1, the latency of accessing different levels of the memory hierarchy varies greatly, and accessing a higher-level cache can be orders of magnitude faster than accessing a lower-level cache or the main memory. As we will show in Sec. 3, when processing a single document, the random accesses of previous LDA algorithms spread across either an $O(KV)$ topic-word count matrix or an $O(DK)$ document-topic count matrix, where D is the number of documents and V is the vocabulary size. As K, V and D can all exceed one million in large-scale applications, both matrices exceed tens of gigabytes in size, which is too large to fit in any cache, resulting in unsatisfactory memory efficiency. Moreover, this size is difficult to reduce for CGS because both matrices need to be accessed for sampling a single token.

In this paper, we propose to reduce the latency of random accesses of LDA by reducing the size of the randomly accessed memory. Based on a careful analysis of existing algorithms, we develop WarpLDA[1], a novel sampling algorithm based on Monte-Carlo Expectation Maximization (MCEM) that preserves the $O(1)$ time complexity per-token and has some carefully designed reordering strategy to achieve an $O(K)$ size of randomly accessed memory per-document, which is small enough to fit in the L3 cache. As the L3 cache is at least six times faster than the main memory (See Table 1), this results in a significant performance gain. Another nice property of WarpLDA is that it simplifies the system design. We present an implementation of WarpLDA in a distributed

[1]The name comes after Warp Drive, the faster-than-light propulsion system in Star Trek.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.
Proceedings of the VLDB Endowment, Vol. 9, No. 10
Copyright 2016 VLDB Endowment 2150-8097/16/06.

and memory-efficient way, with a carefully designed computational framework based on distributed sparse matrices.

Extensive empirical studies in a wide range of settings demonstrate that WarpLDA consistently converges 5-15x faster than the state-of-the-art Metropolis-Hastings based LightLDA [32] and in most settings, faster than the state-of-the-art sparsity aware F+-LDA [31] for learning LDA models with thousands to millions topics. WarpLDA scales to corpora of billion-scale documents, and achieves an unprecedented 11 billion tokens per second throughput with 256 machines.

Outline: Section 2 introduces some basics of LDA, Section 3 provides the memory efficiency analysis of existing algorithms. Section 4 introduces the WarpLDA algorithm, and Section 5 provides system design and implementation details. Experiments are presented in Section 6. Section 7 concludes.

2. BACKGROUNDS

2.1 Basics of LDA

Let $\mathbf{w}_d = \{w_{dn}\}_{n=1}^{L_d}$ denote the collection of L_d words in document d and $\mathbf{W} = \{\mathbf{w}_d\}_{d=1}^{D}$ be a corpus of D documents. Let V denote the vocabulary size. Latent Dirichlet Allocation [7] is a hierarchical Bayesian model, which models the distribution of a word as a mixture of K topic distributions, with the shared mixing proportion for all the words within the same document. Formally, each topic k is a V-dim word distribution $\boldsymbol{\phi}_k$, which follows a Dirichlet prior $\boldsymbol{\phi}_k \sim \text{Dir}(\beta\mathbf{1})$ with parameter β; and the generative process of LDA for each document d is:

Draw a K-dim topic mixing proportion: $\boldsymbol{\theta}_d \sim \text{Dir}(\boldsymbol{\alpha})$,

For each position $n \in \{1, \dots, L_d\}$:

Draw a topic assignment: $z_{dn} \sim \text{Mult}(\boldsymbol{\theta}_d)$,

Draw a word: $w_{dn} \sim \text{Mult}(\boldsymbol{\phi}_{z_{dn}})$,

where $\text{Mult}(\cdot)$ is a multinomial distribution (or categorical distribution), and $\boldsymbol{\alpha}$ are Dirichlet parameters. Let $\boldsymbol{\Phi} = [\boldsymbol{\phi}_1 \cdots \boldsymbol{\phi}_K]$ be the $K \times V$ topic matrix. We further denote $\boldsymbol{\Theta} = \{\boldsymbol{\theta}_d\}_{d=1}^{D}$ and $\mathbf{Z} = \{\mathbf{z}_d\}_{d=1}^{D}$, where $\mathbf{z}_d = \{z_{dn}\}_{n=1}^{L_d}$. Let $\bar{\alpha} = \sum_{k=1}^{K} \alpha_k$ and $\bar{\beta} = V\beta$. In this paper, we define *token* as an occurrence of a *word*, e.g., "apple" is a word, and each of its occurrence is a token. When we say "all tokens of word w" we mean all the occurrences of w, which may have different topic assignments and we use $\mathbf{z}_w$ to denote the topic assignments of all tokens of word w.

To train LDA, one must infer the posterior distribution (or its marginal version) of latent variables $(\boldsymbol{\Theta}, \boldsymbol{\Phi}, \mathbf{Z})$ given $(\mathbf{W}, \boldsymbol{\alpha}, \beta)$. Unfortunately, exact posterior inference is intractable. Thus approximate techniques including variational Bayes and Markov Chain Monte Carlo (MCMC) methods are adopted. As mentioned before, Collapsed Gibbs Sampling (CGS) [16] is most popular because of its simplicity and the availability of fast sampling algorithms. Given $(\mathbf{W}, \boldsymbol{\alpha}, \beta)$, CGS integrates out $(\boldsymbol{\Theta}, \boldsymbol{\Phi})$ by conjugacy and iteratively samples z_{dn} from the local conditional distribution:

$$p(z_{dn}{=}k|\mathbf{Z}_{\neg dn}, w_{dn}{=}w, \mathbf{W}_{\neg dn}) \propto (C_{dk}^{\neg dn}+\alpha_k)\frac{C_{wk}^{\neg dn}+\beta}{C_k^{\neg dn}+\bar{\beta}}, \quad (1)$$

where $C_{dk} = \sum_{n=1}^{L_d} \mathbb{I}(z_{dn} = k)$ is the number of tokens that are assigned to topic k in document d; $C_{wk} = \sum_{d=1}^{D}\sum_{n=1}^{L_d} \mathbb{I}(z_{dn} = k, w_{dn} = w)$ is the number of times that word w has topic k; $C_k = \sum_d C_{dk} = \sum_w C_{wk}$. [2] The superscript or subscript $^{\neg dn}$ stands for excluding (z_{dn}, w_{dn}) from the corresponding count or collection. We further define $\mathbf{C}_d$ to be the $D \times K$ matrix formed by C_{dk}, and $\mathbf{C}_w$ to be the $V \times K$ matrix formed by C_{wk}, with $\mathbf{c}_d$ and $\mathbf{c}_w$ being their particular rows indexed by d and w, and K_d, K_w being the number of non-zero entries of the corresponding row. Let the global topic count vector $\mathbf{c}_k = (C_1, \dots, C_K)^\top$.

By sampling in a collapsed space, CGS often converges faster than a standard Gibbs sampler in a space with all the variables. A straightforward implementation of Eq. (1) is of complexity $O(K)$ per-token by naively enumerating all the K possible topic assignments. This can be too expensive for large-scale applications where K can be in the order of 10^6. Various fast sampling algorithms [30, 20, 32, 31] exist to speed this up, as we shall see in Sec. 3.

[2] We distinguish different counts by their subscripts.

Algorithm 1 Metropolis-Hastings algorithm

Require: Initial state x_0, $p(x)$, $q(\hat{x}|x)$, number of steps M

for $t \leftarrow 1$ to M **do**

Draw $\hat{x} \sim q(\hat{x}|x_{t-1})$

Compute the acceptance rate $\pi = \min\{1, \frac{p(\hat{x})q(x_{t-1}|\hat{x})}{p(x_{t-1})q(\hat{x}|x_{t-1})}\}$

$$x_t = \begin{cases} \hat{x} & \text{with probability } \pi \\ x_{t-1} & \text{otherwise} \end{cases}$$

end for

2.2 Sample from Probability Distributions

Before introducing the algorithm, we first describe some useful tools to sample from probability distributions, which are used by WarpLDA and other algorithms [32, 20].

Metropolis-Hastings (MH): Let $p(x)$ be an (unnormalized) target distribution. We consider the nontrivial case that it is hard to directly draw samples from $p(x)$. MH methods construct a Markov chain with an easy-to-sample *proposal distribution* $q(\hat{x}_t|x_{t-1})$ at each step t. Starting with an arbitrary state x_0, MH repeatedly generates samples from the proposal distribution $\hat{x}_t \sim q(\hat{x}_t|x_{t-1})$, and updates the current state with the new sample with an *acceptance rate* $\pi_t = \min\{1, \frac{p(\hat{x}_t)q(x_{t-1}|\hat{x}_t)}{p(x_{t-1})q(\hat{x}_t|x_{t-1})}\}$. Under some mild conditions, it is guaranteed that $p(x_t)$ converges to $p(x)$ as $t \to \infty$, regardless of x_0 and $q(\hat{x}|x)$ (See Alg. 1). In LDA, $p(x)$ is the distribution of topic assignment in Eq. (1), whose sampling complexity is $O(K)$, and $q(\hat{x}_t|x_{t-1})$ is often a cheap approximation of $p(x)$, as will be clear soon.

Mixture of multinomials: If a multinomial distribution has the form

$$p(x = k) \propto A_k + B_k,$$

it can be represented by a mixture of two distributions,

$$p(x = k) = \frac{Z_A}{Z_A + Z_B}p_A(x = k) + \frac{Z_B}{Z_A + Z_B}p_B(x = k),$$

where $Z_A = \sum_k A_k$ and $Z_B = \sum_k B_k$ are the normalizing coefficients, and $p_A(x = k) = \frac{A_k}{Z_A}, p_B(x = k) = \frac{B_k}{Z_B}$ are the normalized mixture distributions. By introducing an extra binary variable u, which follows a Bernoulli distribution $\text{Bern}(\frac{Z_A}{Z_A+Z_B})$, and defining $p(x|u = 1) = p_A(x), p(x|u = 0) = p_B(x)$, one can confirm that $p(x)$ is a marginal distribution of $p(u)p(x|u)$. Therefore, a sample from $p(x)$ can be drawn via an ancestral sampler, which first draws $u \sim \text{Bern}(\frac{Z_A}{Z_A+Z_B})$ and then samples x from $p_A(x)$ if $u = 1$ and from $p_B(x)$ if $u = 0$. This principle is useful when both $p_A(x)$ and $p_B(x)$ are easy to sample from.

Alias Sampling: Alias sampling [27] is a technique to draw samples from a K-dim multinomial distribution in $O(1)$ after $O(K)$ construction of an auxiliary structure called alias table. The alias table has K bins with equal probability, with at most two outcomes

Table 2: Existing algorithms of LDA, where K: number of topics, K_d: average number of topics per-document, K_w: average number of topics per word, D: number of documents, V: size of vocabulary, SA: sparsity-aware algorithm, MH: MH-based algorithm.

Algorithm	Type	Amount of sequential accesses O(per-token)	Number of random accesses O(per-token)	Size of randomly accessed memory per-document	Order
CGS [16]	-	K	-	-	doc
SparseLDA [30]	SA	$K_d + K_w$	$K_d + K_w$	KV	doc
AliasLDA [20]	SA&MH	K_d	K_d	KV	doc
F+LDA [31]	SA	K_d	K_d	DK	word
LightLDA [32]	MH	-	1	KV	doc
WarpLDA	MH	-	**1**	$\boldsymbol{K}$	doc&word

in each bin. The samples can be obtained by randomly selecting a bin and then randomly selecting an outcome from that bin. Alias sampling has been used in previous LDA algorithms [20, 32].

3. ANALYSIS OF EXISTING ALGORITHMS

We now describe our methodology to analyze the memory access efficiency, and analyze of existing fast sampling algorithms of LDA [30, 21, 32, 31] to motivate the development of WarpLDA.

3.1 Methodology

The running time of LDA is often dominated by random memory accesses, whose efficiency depends on the average *latency* of each access. As accessing main memory is very slow, modern computers exploit *locality* to reduce the latency. If many memory accesses are concentrated in a small area which fits in the cache, the latency will reduce greatly, as shown in Table 1. We focus on the L3 cache in this paper, which is about 30 megabytes in size, and is at least six times faster than the main memory.

The size of randomly accessed memory is an important factor to the latency, because a smaller memory region can fit in a higher-level cache whose latency is lower. So we analyze the memory access efficiency by the size of randomly accessed memory, more precisely, *the size of possibly accessed memory region when sampling the topic assignments for a document $\mathbf{z}_d$ or a word $\mathbf{z}_w$*, depending on whether the sampling algorithm visits the tokens *document-by-document* or *word-by-word*, which will be defined soon.

As we shall see soon, there are two main types of random access encountered for LDA: (1) randomly accessing a matrix of size $O(KV)$ or $O(DK)$; and (2) randomly accessing a vector of size $O(K)$. As K, V and D can all exceed one million, the matrix is typically tens of gigabytes in size even when it is sparse; but the vector is megabytes or smaller in size, and fits in the L3 cache. Therefore, it is reasonable to assume that random accesses to matrices are not efficient, while random accesses to vectors are efficient.

Existing fast sampling algorithms for LDA follow CGS that we introduced in Sec. 2.1, which iteratively visits every token of the corpus, and samples the topic assignments based on the count matrices $\mathbf{C}_d$ and $\mathbf{C}_w$. The tokens can be visited in any order. Two commonly used ordering are *document-by-document* which firstly visits all tokens for document 1, and then visits all tokens for document 2, and so on; and *word-by-word* which firstly visits all tokens for word 1, and then visits all tokens for word 2, and so on. These orderings determine which one of the two count matrices $\mathbf{C}_d$ and $\mathbf{C}_w$ can be accessed efficiently, as we will see in Sec. 3.3.

3.2 Existing Fast Sampling Algorithms

We now summarize existing fast sampling algorithms [30, 20, 32, 31]. These algorithms can be categorized as being either sparsity-aware or MH-based, and they have different factorizations to the basic CGS sampling formula Eq. (1). For clarity all the $^{\neg dn}$ superscripts of CGS based algorithms are omitted.

Sparsity-aware algorithms utilize the sparse structure of the count matrices $\mathbf{C}_d$ and $\mathbf{C}_w$. For example, SparseLDA [30] has the factorization $C_{wk}\frac{C_{dk}+\alpha_k}{C_k+\beta} + \beta\frac{C_{dk}}{C_k+\beta} + \frac{\alpha_k\beta}{C_k+\beta}$ and it enumerates all non-zero entries of $\mathbf{c}_w$ and $\mathbf{c}_d$ to calculate the normalizing constant of each term, which are $\sum_{k=1}^K C_{wk}\frac{C_{dk}+\alpha_k}{C_k+\beta}$, $\sum_{k=1}^K \beta\frac{C_{dk}}{C_k+\beta}$ and $\sum_{k=1}^K \frac{\alpha_k\beta}{C_k+\beta}$ respectively. AliasLDA [20] has the factorization $C_{dk}\frac{C_{wk}+\beta}{C_k+\beta} + \alpha_k\frac{C_{wk}+\beta}{C_k+\beta}$, where C_{wk} and C_k in the latter term are approximated with their stale versions. AliasLDA enumerates all non-zero entries of $\mathbf{c}_d$ to calculate the normalizing constant of the first term, and an alias table is used to draw samples from the latter term in amortized $O(1)$ time. Then, an MH step is used to correct the bias of stale topic counts. F+LDA [31] has the same factorization as AliasLDA but visits the tokens word-by-word, and use a F+ tree for the exact sampling of the latter term.

MH-based algorithms rely on some easy-to-sample proposal distributions to explore the state space $\{1, \dots, K\}$. The proposal distributions are not necessarily sparse, and hence MH-based algorithms can be applied to models whose $p(z_{dn})$'s do not have sparsity structures, e.g., MedLDA [35] or dynamic topic models [4]. For example, LightLDA [32] alternatively draws samples from two simple proposal distributions $q^{\text{doc}} \propto C_{dk} + \alpha_k$ and $q^{\text{word}} \propto \frac{C_{wk}+\beta}{C_k+\beta}$, and accepts the proposals by the corresponding acceptance rate. The time complexity of sampling for a token is $O(1)$. The $O(1)$ complexity of LightLDA has already reached the theoretical lower bound, however its practical throughput is only roughly 4M tokens per second per machine [32], due to its slow random access.

3.3 Analysis

All the aforementioned algorithms are not memory efficient because they have random accesses to large matrices. Specifically, the main random accesses of these algorithms are to the count matrices $\mathbf{C}_d$ and $\mathbf{C}_w$. Ignoring all details of computing, we only focus on the reading and writing to the count matrices $\mathbf{C}_d$ and $\mathbf{C}_w$. When sampling z_{dn} for each token w_{dn}, the memory access pattern to the count matrices can be summarized as follows:

read C_{dk} and C_{wk}, for all $k \in \mathcal{K}_{dn}$; update z_{dn};

write $C_{dz_{dn}}$ and $C_{w_{dn}z_{dn}}$,

where $k \in \mathcal{K}_{dn}$ is a set. The existing algorithms differ in two aspects: (1) *Ordering of visiting tokens*: SparseLDA, AliasLDA and LightLDA visit tokens document-by-document, while F+LDA visits tokens word-by-word; and (2) *Set $\mathcal{K}_{dn}$*: The set $\mathcal{K}_{dn}$ depends on the sparsity structure of the proposal distribution. For SparseLDA, $\mathcal{K}_{dn}$ is the set of non-zero topics of $\mathbf{c}_w$ and $\mathbf{c}_d$; for AliasLDA and F+LDA, $\mathcal{K}_{dn}$ is the set of non-zero topics of $\mathbf{c}_d$; and for LightLDA, $\mathcal{K}_{dn}$ is the set of some samples from the proposal distribution.

We have two important observations, where the accesses can be made efficient if only $\mathbf{C}_d$ or $\mathbf{C}_w$ is accessed, while the accesses are inefficient if both $\mathbf{C}_d$ and $\mathbf{C}_w$ are accessed, as detailed below:

Accessing either $\mathbf{C}_d$ or $\mathbf{C}_w$: Without loss of generality, assume only $\mathbf{C}_d$ is accessed. There will be lots of accesses to random entries of the matrix $\mathbf{C}_d$. However, these random accesses are sorted by d, if the tokens are visited document-by-document. For sampling the tokens in document d, only one row $\mathbf{c}_d$ is accessed. Therefore, the size of randomly accessed memory per-document is only K_d, which is the size of $\mathbf{c}_d$, and fits in the L3 cache. Moreover, the rows except $\mathbf{c}_d$ need not to be stored in the memory and can be computed on-the-fly upon request, reducing the storage overhead. Symmetrically, if only $\mathbf{C}_w$ is accessed, the accesses can be restricted in a vector $\mathbf{c}_w$ by visiting the tokens word-by-word. The counts $\mathbf{c}_w$ can be computed on-the-fly as well.

Accessing both $\mathbf{C}_d$ and $\mathbf{C}_w$: Unfortunately, the accesses to $\mathbf{C}_d$ and $\mathbf{C}_w$ are coupled in all the aforementioned algorithms, i.e., they need to access both $\mathbf{C}_d$ and $\mathbf{C}_w$ when sampling a token. If the tokens are visited document-by-document, the accesses to $\mathbf{C}_w$ are not sorted by w, and spread in the large matrix $\mathbf{C}_w$ whose size is $O(KV)$. If the tokens are visited word-by-word, the accesses to $\mathbf{C}_d$ are not sorted by d, and spread in the large matrix $\mathbf{C}_d$ whose size is $O(DK)$. Thus, all the existing algorithms have $O(KV)$ or $O(DK)$ size of randomly accessed memory, which is not efficient.

Table 2 summarizes the existing algorithms. Note that the amount of sequential accesses is smaller than or equals to the amount of random accesses for all the fast sampling algorithms. This justifies our previous argument that the running time of LDA is dominated by random accesses based on the fact that sequential accesses are faster than random accesses.

WarpLDA addresses the inefficient memory access problem by decoupling the accesses to $\mathbf{C}_d$ and $\mathbf{C}_w$. Particularly, WarpLDA first visits all tokens but only accesses $\mathbf{C}_d$, and then visits all tokens again but only accesses $\mathbf{C}_w$. As we just showed, we can choose the corresponding ordering of visiting the tokens so that the accesses to both matrices can be restricted in the current row $\mathbf{c}_d$ and $\mathbf{c}_w$, without affecting the correctness of the algorithm.

4. WARPLDA

We now present WarpLDA, a novel MH-based algorithm that finds a maximum *a posteriori* (MAP) estimate of LDA with an $O(1)$ per-token sampling complexity and an $O(K)$ size of randomly accessed memory per-document (or word), by a designed reordering strategy to decouple the accesses to $\mathbf{C}_d$ and $\mathbf{C}_w$.

4.1 Notations

We first define some notations to make presentation clear. Specifically, we define a $D \times V$ *topic assignment matrix* $\mathbf{X}$, where the topic assignment z_{dn} is put in the cell (d, w_{dn}) (See Fig. 1 for an illustration). Note that there might be multiple topic assignments in a single cell because a word can appear more than once in a document. Let $\mathbf{z}_d$ be the collection of the L_d topic assignments in the d-th row of $\mathbf{X}$, with z_{dn} being its n-th element. The ordering of the elements in $\mathbf{z}_d$ can be arbitrary because LDA is a bag-of-words model and ignores word ordering. This definition of z_{dn} is consistent with that in Sec. 2.1, where z_{dn} is the topic assignment of the n-th token of document d. Again, let $\mathbf{Z} = \mathbf{Z}_d = \{\mathbf{z}_d\}_{d=1}^D$. Similarly, let $\mathbf{z}_w$ be the collection of the topic assignments in the w-th column of $\mathbf{X}$, with size L_w and z_{wn} being its n-th element. Note that L_w is the *term frequency* of word w, i.e., the number of times that w appears in the entire corpus. Let $\mathbf{Z}_w = \{\mathbf{z}_w\}_{w=1}^V$.

Besides a topic assignment, WarpLDA has M topic proposals for each token, denoted as $z_{dn}^{(i)}$ (or $z_{wn}^{(i)}$), where $i = 1, \dots, M$. Using the above definition, we have $\mathbf{z}_d^{(i)}$ (or $\mathbf{z}_w^{(i)}$) for a document and $\mathbf{Z}_d^{(i)}$ (or $\mathbf{Z}_w^{(i)}$) for the whole corpus.

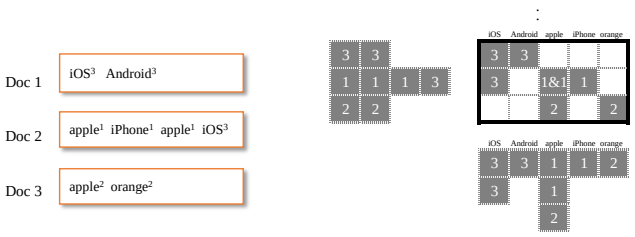


Figure 1: Data representation of WarpLDA. Left: documents, the subscript of each token is its topic assignment z_{dn}. Right: the topic assignment matrix $\mathbf{X}$, $\mathbf{Z}_d$ and $\mathbf{Z}_w$. The & sign separates multiple entries in one cell.

It is worth noting that most of these notations are for the ease of presentation, and in the actual implementation we only store $\mathbf{Z}_w, \mathbf{Z}_w^{(i)}$ and the global count vector $\mathbf{c}_k$ (Sec. 5). The other variables, including $\mathbf{X}, \mathbf{Z}_d, \mathbf{C}_d, \mathbf{C}_w$ are either *views* of $(\mathbf{Z}_w, \mathbf{Z}_w^{(i)})$, or can be computed on-the-fly.

4.2 MCEM Algorithm of LDA

As analysed above, in the original CGS Eq. (1) and existing fast algorithms, it is difficult to decouple the access to $\mathbf{C}_d$ and $\mathbf{C}_w$, because both counts need to be updated instantly after the sampling of every token. We develop our efficient WarpLDA based on a new Monte-Carlo Expectation Maximization (MCEM) algorithm which is similar with CGS, but both counts are fixed until the sampling of all tokens are finished. This scheme allows us to develop a reordering strategy to decouple the accesses to $\mathbf{C}_d$ and $\mathbf{C}_w$, and minimize the size of randomly accessed memory.

Specifically, WarpLDA seeks an MAP solution of the latent variables $\boldsymbol{\Theta}$ and $\boldsymbol{\Phi}$, with the latent topic assignments $\mathbf{Z}$ integrated out:

$$\hat{\boldsymbol{\Theta}}, \hat{\boldsymbol{\Phi}} = \operatorname*{argmax}_{\boldsymbol{\Theta}, \boldsymbol{\Phi}} \log p(\boldsymbol{\Theta}, \boldsymbol{\Phi} | \mathbf{W}, \boldsymbol{\alpha}', \beta'),$$

where $\boldsymbol{\alpha}'$ and β' are the Dirichlet hyper-parameters. Asuncion et al. [3] have shown that this MAP solution is almost identical with the solution of CGS, with proper hyper-parameters. Our empirical results in Sec. 6.3 also support this conclusion.

Computing $\log p(\boldsymbol{\Theta}, \boldsymbol{\Phi} | \mathbf{W}, \boldsymbol{\alpha}', \beta')$ directly is expensive because it needs to enumerate all the K possible topic assignments for each token. We therefore optimize its lower bound as a surrogate. Let $q(\mathbf{Z})$ be a variational distribution. Then, by Jensen's inequality, we get the lower bound $\mathcal{J}(\boldsymbol{\Theta}, \boldsymbol{\Phi}, q(\mathbf{Z}))$:

$$\begin{aligned} \log p(\boldsymbol{\Theta}, \boldsymbol{\Phi} | \mathbf{W}, \boldsymbol{\alpha}', \beta') \geq & \mathbb{E}_q[\log p(\mathbf{W}, \mathbf{Z} | \boldsymbol{\Theta}, \boldsymbol{\Phi}) - \log q(\mathbf{Z})] \\ & + \log p(\boldsymbol{\Theta} | \boldsymbol{\alpha}') + \log p(\boldsymbol{\Phi} | \beta') \\ \triangleq & \mathcal{J}(\boldsymbol{\Theta}, \boldsymbol{\Phi}, q(\mathbf{Z})). \end{aligned} \tag{2}$$

We then develop an Expectation Maximization (EM) algorithm [15] to find a local maximum of the posterior $p(\boldsymbol{\Theta}, \boldsymbol{\Phi} | \mathbf{W}, \boldsymbol{\alpha}', \beta')$, where the E-step maximizes $\mathcal{J}$ with respect to the variational distribution $q(\mathbf{Z})$ and the M-step maximizes $\mathcal{J}$ with respect the to model parameters $(\boldsymbol{\Theta}, \boldsymbol{\Phi})$, while keeping $q(\mathbf{Z})$ fixed. One can prove that the optimal solution at E-step is $q(\mathbf{Z}) = p(\mathbf{Z} | \mathbf{W}, \boldsymbol{\Theta}, \boldsymbol{\Phi})$ without further assumption on q. We apply Monte-Carlo approximation on the expectation in Eq. (2),

$$\mathbb{E}_q[\log p(\mathbf{W}, \mathbf{Z} | \boldsymbol{\Theta}, \boldsymbol{\Phi}) - \log q(\mathbf{Z})] \approx$$

$$\frac{1}{S} \sum_{s=1}^{S} \log p(\mathbf{W}, \mathbf{Z}^{(s)} | \boldsymbol{\Theta}, \boldsymbol{\Phi}) - \log q(\mathbf{Z}^{(s)}),$$

where $\mathbf{Z}^{(1)}, \dots, \mathbf{Z}^{(S)} \sim q(\mathbf{Z}) = p(\mathbf{Z}|\mathbf{W}, \boldsymbol{\Theta}, \boldsymbol{\Phi})$. We set the sample size $S = 1$ and use $\mathbf{Z}$ as an abbreviation of $\mathbf{Z}^{(1)}$.

Sampling Z: Each dimension of $\mathbf{Z}$ can be sampled independently,

$$q(z_{dn} = k) \propto p(\mathbf{W}, \mathbf{Z}|\boldsymbol{\Theta}, \boldsymbol{\Phi}) \propto \theta_{dk}\phi_{w_{dn},k}. \tag{3}$$

Optimizing $\boldsymbol{\Theta}, \boldsymbol{\Phi}$: With the Monte-Carlo approximation,

$$\begin{aligned}\mathcal{J} &\approx \log p(\mathbf{W}, \mathbf{Z}|\boldsymbol{\Theta}, \boldsymbol{\Phi}) + \log p(\boldsymbol{\Theta}|\boldsymbol{\alpha}') + \log p(\boldsymbol{\Phi}|\beta') + \text{const.} \\ &= \sum_{d,k}(C_{dk} + \alpha'_k - 1)\log\theta_{dk} \\ &+ \sum_{k,w}(C_{kw} + \beta' - 1)\log\phi_{kw} + \text{const.},\end{aligned}$$

with the optimal solutions

$$\hat{\theta}_{dk} \propto C_{dk} + \alpha'_k - 1, \quad \hat{\phi}_{wk} = \frac{C_{wk} + \beta' - 1}{C_k + \bar{\beta}' - V}. \tag{4}$$

Instead of computing and storing $\hat{\boldsymbol{\Theta}}$ and $\hat{\boldsymbol{\Phi}}$, we compute and store $\mathbf{C}_d$ and $\mathbf{C}_w$ to save memory because the latter are sparse. Plug Eq. (4) to Eq. (3), and let $\boldsymbol{\alpha} = \boldsymbol{\alpha}' - 1, \beta = \beta' - 1$, we get the final MCEM algorithm, which iteratively performs the following two steps until a given iteration number is reached:

- E-step: Sample $z_{dn} \sim q(z_{dn} = k)$, where

$$q(z_{dn} = k) \propto (C_{dk} + \alpha_k)\frac{C_{wk} + \beta_w}{C_k + \bar{\beta}}. \tag{5}$$

- M-step: Compute $\mathbf{C}_d$ and $\mathbf{C}_w$ by $\mathbf{Z}$.

Note the resemblance between Eq. (5) and Eq. (1) intuitively justifies why MCEM leads to similar results with CGS. The difference between MCEM and CGS is that MCEM updates the counts $\mathbf{C}_d$ and $\mathbf{C}_w$ after sampling all z_{dn}s, while CGS updates the counts instantly after sampling each z_{dn}. The strategy that MCEM updates the counts after sampling all z_{dn}s is called *delayed count update*, or simply *delayed update*. MCEM can be viewed as a CGS with delayed update, which has been widely used in existing algorithms [24, 1]. While previous work uses delayed update as a trick, we hereby present a theoretical guarantee to converge to a MAP solution. Delayed update is important for us to decouple the accesses of $\mathbf{C}_d$ and $\mathbf{C}_w$ to improve cache locality, without affecting the correctness, as will be explained in Sec. 4.4. This MCEM formulation is also used to build embarrassingly parallel AliasLDA [33].

4.3 Sampling the Topic Assignment

A naive application of Eq. (5) is $O(K)$ per-token. We now develop an MH algorithm for faster sampling.

Specifically, starting from an initial state $z^{(0)}_{dn}$, we draw samples alternatively from one of the two proposal distributions:

$$\begin{aligned}q^{\text{doc}}(z_{dn} = k) &\propto C_{dk} + \alpha_k \\ q^{\text{word}}(z_{dn} = k) &\propto C_{wk} + \beta,\end{aligned} \tag{6}$$

and update the current state with the acceptance rates:

$$\begin{aligned}\pi^{\text{doc}}_{k\to k'} &= \min\left\{1, \frac{C_{wk'} + \beta}{C_{wk} + \beta}\frac{C_k + \bar{\beta}}{C_{k'} + \bar{\beta}}\right\} \\ \pi^{\text{word}}_{k\to k'} &= \min\left\{1, \frac{C_{dk'} + \alpha_{k'}}{C_{dk} + \alpha_k}\frac{C_k + \bar{\beta}}{C_{k'} + \bar{\beta}}\right\}.\end{aligned} \tag{7}$$

Similar as in Yuan et al.'s work [32], we can prove that this scheme converges to the correct stationary distribution in Eq. (5).

Computing the acceptance rates only involves constant number of arithmetic operations, so it is $O(1)$. The proposal distributions in Eq. (6) are mixture of multinomials mentioned in Section 2.2, with the mixing coefficient $\frac{L_d}{L_d + \bar{\alpha}}$.[3] There are two possible methods to draw samples from $p_A(z_{dn} = k) \propto C_{dk}$ in $O(1)$ time: (1) **Alias sampling**: Build a K_d-dim alias table for all the non-zero entries of $\mathbf{c}_d$; and (2) **Random positioning**: Noticing that $\mathbf{c}_d$ is the count of $\mathbf{z}_d$, randomly select a position $u \in \{1, \dots, L_d\}$, and return z_{du}. Alias sampling is also used to draw samples from $p_B(z_{dn} = k) \propto a_k$ in $O(1)$. Because both sampling from proposal distributions and computing acceptance rates can be done in $O(1)$, the algorithm is $O(1)$ per-token.

4.4 Reordering the Computation

As discussed above, the size of randomly accessed memory per-document or word is an important factor that influences the efficiency, but it is difficult to reduce for CGS due to the coupling of the counts $\mathbf{C}_w$ and $\mathbf{C}_d$. Thanks to the delayed update strategy in MCEM, we are able to decouple the access to $\mathbf{C}_w$ and $\mathbf{C}_d$, and minimize the size of randomly accessed memory via a reordering strategy. Below, we explain how to do the reordering in the E-step and the M-step in turn.

E-step: The E-step samples the topic assignment z_{dn} for each token, while keeping the counts $\mathbf{C}_d, \mathbf{C}_w$ and $\mathbf{c}_k$ fixed. Consider the sampling of a single topic assignment z_{dn} with the MH algorithm. For simplicity, we only consider the document proposal q^{doc}. According to the MH Alg. 1, starting with the initial state z_{dn}, we do the following in the i-th step ($i = 1, \dots, M$):

- Draw the topic proposals $z^{(i)}_{dn}$ according to Eq. (6), where $q^{\text{doc}}(z^{(i)}_{dn} = k) \propto C_{dk} + \alpha_k$;
- Update the current state z_{dn} by the proposal $z^{(i)}_{dn}$, with the probability π, where $\pi = \min\{1, \frac{C_{wk'} + \beta}{C_{wk} + \beta}\frac{C_k + \bar{\beta}}{C_{k'} + \bar{\beta}}\}$, according to Eq. (7).

Both $\mathbf{C}_d$ and $\mathbf{C}_w$ need to be accessed to sample z_{dn} in the above procedure. Following our analysis in Sec. 3.3, there are inevitable random accesses to matrices no matter whether we visit the tokens document-by-document or word-by-word. Thanks to the delayed update strategy in MCEM, we can address this problem by a reordering strategy.

Delayed update makes $\mathbf{C}_d$, $\mathbf{C}_w$ and $\mathbf{c}_k$ fixed during the entire E-step. Subsequently, the proposal distribution q^{doc}, which depends solely on $\mathbf{C}_d$ and $\boldsymbol{\alpha}$, is fixed during the E-step. Therefore, we can draw the topic proposals $z^{(i)}_{dn}$ at any time within the E-step, without affecting the correctness. Particularly, we choose to draw the proposals for all tokens *before computing any acceptance rate*. With this particular ordering, the sampling of all the topic assignments can be done in two separate steps:

1. Draw the topic proposals $z^{(i)}_{dn}$ for all tokens. This only accesses $\mathbf{C}_d$ and $\boldsymbol{\alpha}$.
2. Compute the acceptance rates and update the topic assignments for all tokens. This only accesses $\mathbf{C}_w$, $\mathbf{c}_k$ and β.

Since each step only accesses $\mathbf{C}_d$ or $\mathbf{C}_w$, following the analysis in Sec. 3.3, we can make both steps memory efficient by carefully choosing the ordering of visiting tokens. In the first step, tokens are visited document-by-document. Therefore, when processing the d-th document, only a small vector $\mathbf{c}_d$ for the current document is randomly accessed. In the second step, the tokens are visited word-by-word, and when processing the w-th word, only a small vector $\mathbf{c}_w$ is randomly accessed. Because the vectors are small enough to fit in the L3 cache, WarpLDA is memory efficient.

[3] Due to the symmetry we only consider q^{doc}.

Word proposal: The word proposal q^{word} can be treated similarly as the doc proposal q^{doc}, by drawing the topic proposals for all tokens before computing any acceptance rate. There are also two separate steps for the sampling of all topic assignments: (1) Draw the topic proposals $z_{wn}^{(i)}$ for all tokens (accesses $\mathbf{C}_w$ and β); and (2) Compute the acceptance rates and update the topic assignments for all tokens (accesses $\mathbf{C}_d$, $\mathbf{c}_k$, $\boldsymbol{\alpha}$ and β). These steps can be done efficiently by doing the first step word-by-word and doing the second step document-by-document.

An WarpLDA iteration first samples the topic assignments for each token using the document proposal, and then samples again using the word proposal, which involves four passes of the tokens (two passes for each proposal). To improve the efficiency, these four passes can be compressed to two passes. The *document phase* visits tokens document-by-document, and do the operations that require $\mathbf{c}_d$, i.e., computing the acceptance rates for the word proposal and drawing samples from the document proposal. Symmetrically, the *word phase* visits tokens word-by-word, computes the acceptance rates for the document proposal, and then draws samples from the word proposal.

M-step: Up to now, we discussed about how to do the sampling of the topic assignments, i.e., the E-step of the MCEM algorithm. The M-step, which updates the counts $\mathbf{C}_d$, $\mathbf{C}_w$ and $\mathbf{c}_k$, need not to be conducted explicitly, because the counts can be computed on-the-fly. The only usage of the vector $\mathbf{c}_d$ is when processing document d in the document phase. Hence, it can be computed by $\mathbf{z}_d$ when processing document d, and discarded after the document is finished. Similarly, the only usage of the row vector $\mathbf{c}_w$ is when processing word w in the word phase, and it can be computed on-the-fly as well. Noticing $\mathbf{c}_k = \sum_d \mathbf{c}_d = \sum_w \mathbf{c}_w$, the count vector $\mathbf{c}_k$ can be accumulated when computing $\mathbf{c}_d$ for each document.

The above facts also imply that we even need not to *store* $\mathbf{C}_w$ and $\mathbf{C}_d$, which simplifies the system design as we shall see in Sec. 5, and again, justifies there are no random accesses to matrices — we do not even store any of the matrices $\mathbf{C}_d$ and $\mathbf{C}_w$.

5. SYSTEM IMPLEMENTATION

WarpLDA not only improves the cache locality but also simplifies the distributed system design for training LDA on hundreds of machines. In previous systems for distributed LDA, including parameter servers (PS) [1, 21] and model parallel systems [28, 32], all workers collaboratively refine a globally shared count matrix $\mathbf{C}_w$. This adds additional complications to the system, such as read/write locks or delta threads. WarpLDA is arguably simpler to implement because its only globally shared object is a small vector $\mathbf{c}_k$ which can be updated and broadcast-ed to each machine in every iteration, and all the other data are local so that they can be processed independently by each worker.

In this section, we discuss the design of the distributed sparse matrix framework for WarpLDA. We then present techniques to implement the framework in a memory efficient fashion. Some application-level optimizations are also discussed.

5.1 Programming Model

We start the presentation of our framework for WarpLDA by abstracting its data structure. In WarpLDA, the only data to manipulate are: 1) local per-token data, where each token w_{dn} is associated with $M+1$ integers $y_{dn} \triangleq (z_{dn}, z_{dn}^{(1)}, \ldots, z_{dn}^{(M)})$, which are the topic assignment and the topic proposals; and 2) global topic count vector $\mathbf{c}_k$. The local per-token data are stored in a $D \times V$ matrix $\mathbf{Y}$, where each token w_{dn} corresponds to an entry at the position (d, w_{dn}) with y_{dn} as its data. The matrix $\mathbf{Y}$ has the same structure with $\mathbf{X}$ in Fig. 1, but is augmented in the sense that each entry stores not only the topic assignment but also M proposals.

```
template <class Data>
interface SparseMatrix {
    // Initialization
    void AddEntry(int r, int c, Data data);
    // Computation
    void VisitByRow(Operation op);
    void VisitByColumn(Operation op);
    // User-defined function
    interface Operation {
        operator () (vector<Data>& data);
    };
};
```

Figure 2: Interface of a distributed sparse matrix.

The main data structure of our framework is the *distributed sparse matrix* $\mathbf{Y}$ of $D \times V$ *cells*, where each cell can have zero, one, or multiple *entries*, corresponding to the tokens in LDA, and each entry stores some *data*, which are y_{dn} for WarpLDA. The structure of the matrix, i.e., the positions of the entries, is fixed, and only the data of the entries are iteratively refined to the solution.

To help readers relate our presentation with the actual implementation, we use C++ notations in this section, e.g., $\mathbf{Y}[d][w]$ is the cell at the d-th row and w-th column of $\mathbf{Y}$. Let $row[d]$ as the collection of all entries in the d-th row, with $row[d][n]$ being its n-th element. Similarly, define the columns $col[w]$, and $col[w][n]$. Let $a.size$ be the size of the collection a, such as $row[d].size$ and $col[d].size$. There are several methods to manipulate the matrix (See Fig. 2):

- `AddEntry`: add an entry to a given cell $\mathbf{Y}[d][w]$ with the data `data`. This is called only at initialization time.
- `VisitByRow`: For each row d, update the data for each entry, given the data of all the entries in the current row $row[d]$, i.e., $row[d] \leftarrow f_r(row[d])$, where $f_r(\cdot)$ is a user defined function.
- `VisitByColumn`: For each column w, update the data for each entry, given the data of all the entries in the current column $col[w]$, i.e., $col[w] \leftarrow f_c(col[w])$, where $f_c(\cdot)$ is another user defined function.

To use the framework, users initialize the matrix via the `AddEntry` method, and then call `VisitByRow` and `VisitByColumn` with user defined functions for a number of times to refine the data of the matrix towards the result.

For WarpLDA, the matrix $\mathbf{Y}$ stores the local per-token data y_{dn}, where each row is a document, and each column is a word. One can verify that the topic assignments of the d-th document $\mathbf{z}_d$ are stored in the d-th row $row[d]$, because $row[d][n] = y_{dn} = (z_{dn}, z_{dn}^{(1)}, \ldots, z_{dn}^{(M)})$. Similarly, $\mathbf{z}_w$ are stored in $col[w]$. We firstly initialize the matrix by adding an entry y_{dn} to the cell $\mathbf{Y}[d][w_{dn}]$ for each token. The document phase can be implemented with a single `VisitByRow`, where for each row (document), $\mathbf{c}_d$ is calculated with the topic assignments in $row[d]$; then the topic assignments in $row[d]$ are updated given the topic proposals and $\mathbf{c}_d$; and finally new proposals are created given new topic assignments. [4] Similarly, the word phase can be implemented by `VisitByColumn`. The globally shared vector $\mathbf{c}_k = \sum_w \mathbf{c}_w$ can be updated by a simple "reduce" operation, which aggregates the word-topic count vectors $\mathbf{c}_w$, in the user defined function $f_c(\cdot)$. Since updating $\mathbf{c}_k$ can

[4] Refer to Alg. 2 in the Appendix for details.

be implemented by the user, our framework does not handle the global count vector $\mathbf{c}_k$ but only $\mathbf{Y}$.

A basic implementation of this framework is MapReduce. Each entry is represented by a $(d, w) \to \mathbf{Y}[d][w]$ pair, and `VisitByRow` is implemented by two steps: 1) take the entries $(d, w) \to \mathbf{Y}[d][w]$, aggregate them by row, and emit the rows $d \to row[d]$; 2) take the rows (documents) $d \to row[d]$, do the update $row[d] \leftarrow f_r(row[d])$ and emit the individual entries $(d, w) \to \mathbf{Y}[d][w]$. `VisitByColumn` is implemented by two similar steps, but aggregate the entries by column. This implementation is useful for industrial users who want to build a simple distributed $O(1)$ LDA on top of the existing MapReduce framework. However, the shuffling overhead of MapReduce is often too large and the performance is unsatisfactory. Hence, we present a dedicated implementation.

5.2 Data Layout

The first problem we need to address is how the matrix is represented internally. `VisitByRow` (or `VisitByColumn`) requires to provide $row[d]$ (or $col[w]$) to the user defined functions, while both $row[d]$ and $col[w]$ should be accessed efficiently, and the change of either should reflect to the underlying matrix.

There are a number of formats for a sparse matrix. Two most well-known examples are the Compressed Sparse Row (CSR) format and the Compressed Sparse Column (CSC) format. In the CSR representation $\mathbf{Y}_{CSR}$, the rows are stored continuously, and in the CSC representation $\mathbf{Y}_{CSC}$, the columns are stored continuously.

One possible data layout is storing both $\mathbf{Y}_{CSR}$ and $\mathbf{Y}_{CSC}$. After each `VisitByRow` or `VisitByColumn` operation, perform a "transpose" operation to synchronize the change of one representation to the other representation. By storing both $\mathbf{Y}_{CSR}$ and $\mathbf{Y}_{CSC}$, the user defined operation always has sequential accesses to the data and hence is memory efficient. However, the transpose operation requires an extra pass of data which is expensive.

In WarpLDA, we avoid explicitly transposing the matrix by only storing $\mathbf{Y}_{CSC}$. For accessing the rows of the matrix, we store $\mathbf{P}_{CSR}$, which are *pointers* to the entries in $\mathbf{Y}_{CSC}$. Since only one copy of the data is stored, there is no need for explicitly transposing the matrix. However, the rows are accessed by indirect memory accesses. We now show these indirect memory accesses are still memory efficient because the cache lines are fully utilized.

For each column $col[w]$, we sort the entries by their row id. Then, while in `VisitByRow`, the entries of a particular column $col[w]$ are always accessed monotonically, i.e., $col[w][i]$ is always accessed before $col[w][j]$ for all $i < j$. When an entry $col[w][i]$ is accessed, the cache line containing it is fetched to the cache, which also contains the next few entries $(col[w][i+1], col[w][i+2], \dots)$. As long as the cache is large enough to store one cache line for each column, the cache lines can stay in the cache until all the entries on it are accessed. Thus, the cache lines are fully utilized. Moreover, the size of the columns follows the power-law, because they are term-frequencies of words in natural corpora [19]. Therefore, the required cache size can be even smaller, comparable to the number of columns which have most entries of the sparse matrix. For example, in the ClueWeb12 corpus, the vocabulary size (number of columns) is 1,000,000, the first 10,000 words (columns) attributes to 80% of the entries, and storing a cache line for these words (columns) requires only $10000 \times 64\text{B} = 625\text{KB}$.

5.3 Scaling out

WarpLDA can be scaled out to hundreds of machines to meet the requirements of learning large models on massive-scale corpora. We now present the key components for scaling out WarpLDA, including task partitioning, data placement and communication.

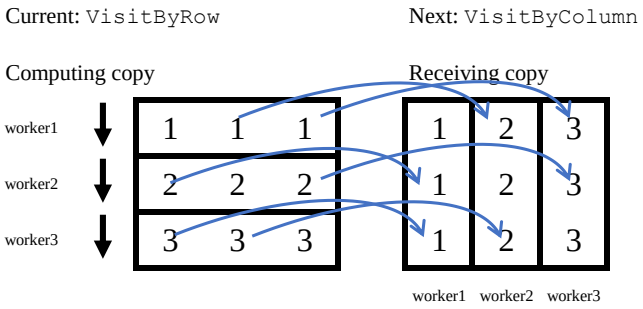


Figure 3: Data partitioning. The number indicates the worker each partition belongs to, the blue arrows are data flows during communication, and the black arrows are the directions of visiting data.

5.3.1 *Multi-threading and NUMA*

WarpLDA is embarrassingly parallel because the workers operate on disjoint sets of data. To parallelize WarpLDA we only need to invoke the user defined functions in parallel for different rows and columns. In contrast, traditional frameworks such as Yahoo!LDA [1] and LightLDA [32] need to update the count matrix $\mathbf{C}_w$ in parallel, and require extra treatments such as read/write locks and delta threads.

Modern computers have non-uniform memory access (NUMA), where each main memory DIMM belongs to a specific CPU socket. If one CPU needs to access data in the memory belongs to another CPU socket, the data flows through another CPU socket, resulting in degraded performance. For better performance, WarpLDA partitions the data Y_{CSC} by column and the pointers P_{CSR} by row, and bind each slice to a different CPU socket. In this way, `VisitByColumn` accesses only the local data, but the indirect memory accesses via P_{CSR} in `VisitByRow` may flow through other CPUs.

5.3.2 *Fine-grained Distributed Computing*

Distributed computing typically involves partitioning the task and addressing the communications between workers. We take a two-level partitioning strategy to overlap the computation and communication, and propose a greedy approach for balanced partitioning on the challenging case where the length of columns are distributed as power-law. The communications are implemented with `MPI_Ialltoall` in Message Passing Interface (MPI).

Each `VisitByRow` and `VisitByColumn` involves a pass of the matrix $\mathbf{Y}$. To assign the task to different workers, we split the $D \times V$ data matrix $\mathbf{Y}$ as $P \times P$ *partitions*, where P is the number of MPI workers, and P_{ij} is the (i, j)-th partition. In `VisitByRow`, the data is partitioned by row, i.e., worker i has partitions $P_{i1}, \dots, P_{iP}$ in its memory; in `VisitByColumn`, the data is partitioned by column, i.e., worker i has partitions $P_{1i}, \dots, P_{Pi}$ in its memory. Exchange of data happens only when the adjacent two operations are different, i.e., one is `VisitByRow` and the other is `VisitByColumn`, in which case the partitions are sent to their corresponding workers of the next operation after the current operation is finished. Because MPI requires both sending buffer and receiving buffer, we maintain a computing copy and receiving copy of the data. For example, in Fig. 3, the current operation is `VisitByRow`, which can be conducted independently on each worker without any communications, due to our partitioning strategy. After the current `VisitByRow` is finished, we need to partition the matrix by column for the next `VisitByColumn`, so we send partition P_{ij} from worker i to worker j, with `MPI_Ialltoall`. If the next operation is `VisitByRow` instead, no communication is needed.

We can overlap the communication and computation by further

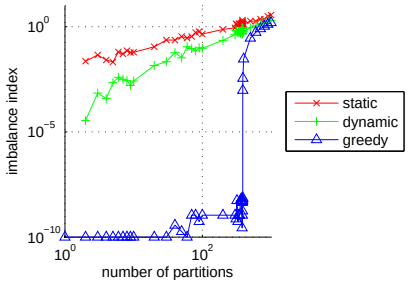


Figure 4: Comparison of partition strategies on ClueWeb12.

divide each partition as $B \times B$ *blocks*, where $B \in [2, 10]$ is a small constant. During training, each of the blocks may be in one of the four states: 1) not started; 2) computing; 3) sending; 4) finished. In each `VisitByRow` / `VisitByColumn`, workers scan their blocks by row or column, and each block can be immediately sent to the receiving copy after it is finished.

To minimize the wait time, the number of tokens within each worker should be roughly the same, which implies that the sum of the number of tokens of all partitions within each row or column should be roughly the same. Note that the rows / columns can be treated independently: we partition the rows as P disjoint sets $R_1, \dots, R_P$ and the columns as disjoint sets $C_1, \dots, C_P$, so that each set has roughly the same number of tokens. Given the partitioning of rows and columns, the (i, j)-th partition $P_{ij} = R_i \cap C_j$.

Balanced partitioning of the columns is challenging because the term frequencies of words from a natural corpus, i.e., $col[w].size$, can be very imbalanced because of the power-law [19]. For example, the most frequent word in the ClueWeb12 corpus occupies 0.257% of all occurrences, after the removal of stop words. Considering each slice only have 1% of the tokens if there are 100 slices, this is a very large proportion, and random partitioning can be highly imbalanced in this case. We propose a greedy algorithm for balanced partitioning. First, all the words are sorted by their frequency in a decreasing order. Then from the most frequent word, we put each word (i.e., column) in the slice with the least number of total tokens. Because there are many low frequency words (the long tail), this algorithm can produce very balanced results. We compared the *imbalance index* of our greedy algorithm with two randomized algorithms: *static* first random shuffle the words, then partition so that each partition has equal number of words; *dynamic* allows each partition to have different number of words, but the slices are continuous. Imbalance index is defined as

$$\frac{\text{number of tokens in the largest partition}}{\text{average number of tokens of each partition}} - 1.$$

In the ideal case the imbalance index should be zero. Fig. 4 shows the experimental results on the ClueWeb12 corpus, where we can see that the greedy algorithm is much better than both randomized algorithms. The imbalance index of the greedy algorithm grows dramatically when the number of machines reach a few hundreds, because the size of the largest column is so large that it is impossible to partition the columns in balance.

5.4 Application Level Optimizations

Besides system level optimizations, there are also some application level optimizations (i.e., optimization for the user defined function) for even better performance of WarpLDA. We describe them in this subsection.

Sparse vectors as hash tables: When K is large, it is likely to have $K_d \ll K$ and $K_w \ll K$. It is more effective to use hash tables rather than dense arrays for the counts $\mathbf{c}_d$ and $\mathbf{c}_w$, because the size of a hash table is much smaller than that of a dense array, thus the cost of clearing the counts is smaller, and the size of randomly accessed memory is smaller. We choose an open addressing hash table with linear probing for hash collisions. The hash function is a simple `and` function, and the capacity is set to the minimum power of 2 that is larger than $\min\{K, 2L_d\}$ or $\min\{K, 2L_w\}$. We find that the hash table is almost as fast as a dense array even when $L_d > K$. Although LightLDA [32] also uses hash tables to store the counts, it is mainly for reducing the storage overhead instead of improving cache locality, because its size of randomly accessed memory is too large for the cache anyway.

Intra-word parallelism: For the most frequent words in the corpus, L_w, i.e., the term frequency can be extremely large (e.g., tens of millions), so $L_w \gg K$. It is desirable to exploit *intra-word parallelism* in this case. Making all the threads working at the same column is beneficial for better cache locality, because only the $\mathbf{c}_w$ for *one* word needs to be stored in the cache. Moreover, it helps balancing the load between workers in the case of $col[w].size$ is too large. We choose to exploit intra-word parallelism for words which $L_w > K$. As the user defined function is itself parallel in this case, we let the framework only invoke one user defined function at a time.

To parallelize the user defined function, we compute the count $\mathbf{c}_w$ on each thread, aggregate the results, and construct the alias table by a concurrent vector. Updating the topic assignments is embarrassingly parallel.

Table 3: Statistics of various datasets, where T is the total number of words in the corpus.

Dataset	D	T	V	T/D
NYTimes	300K	100M	102K	332
PubMed	8.2M	738M	141K	90
ClueWeb12 (subset)	38M	14B	1M	367
ClueWeb12	639M	236B	1M	378

6. EXPERIMENTS

We now present empirical studies of WarpLDA, by comparing it with two strong baselines LightLDA [32] and F+LDA [31]. LightLDA is the fastest MH based algorithm, and F+LDA is the fastest sparsity-aware algorithm. We compare with them in both time efficiency and the quality of convergence.

6.1 Datasets and Setups

Table 3 summarizes the datasets. NYTimes and PubMed are standard datasets from the UCI machine learning repository [2]; they consist of news articles and biomedical literature abstracts, respectively. ClueWeb12 is a large crawl of web pages, and ClueWeb12 (subset) is a random subset of the full ClueWeb12.[5]

The experiments are conducted on the Tianhe-2 supercomputer. Each node is equipped with two Xeon E5-2692v2 CPUs (2×12 2.2GHz cores), and 64GB memory. Nodes are connected with InfiniBand, and single machine experiments are done with one node.

We set the hyper-parameters $\alpha = 50/K$ and $\beta = 0.01$. Following the previous work [1, 32], we measure the model quality by the

[5] `http://www.lemurproject.org/clueweb12.php/`

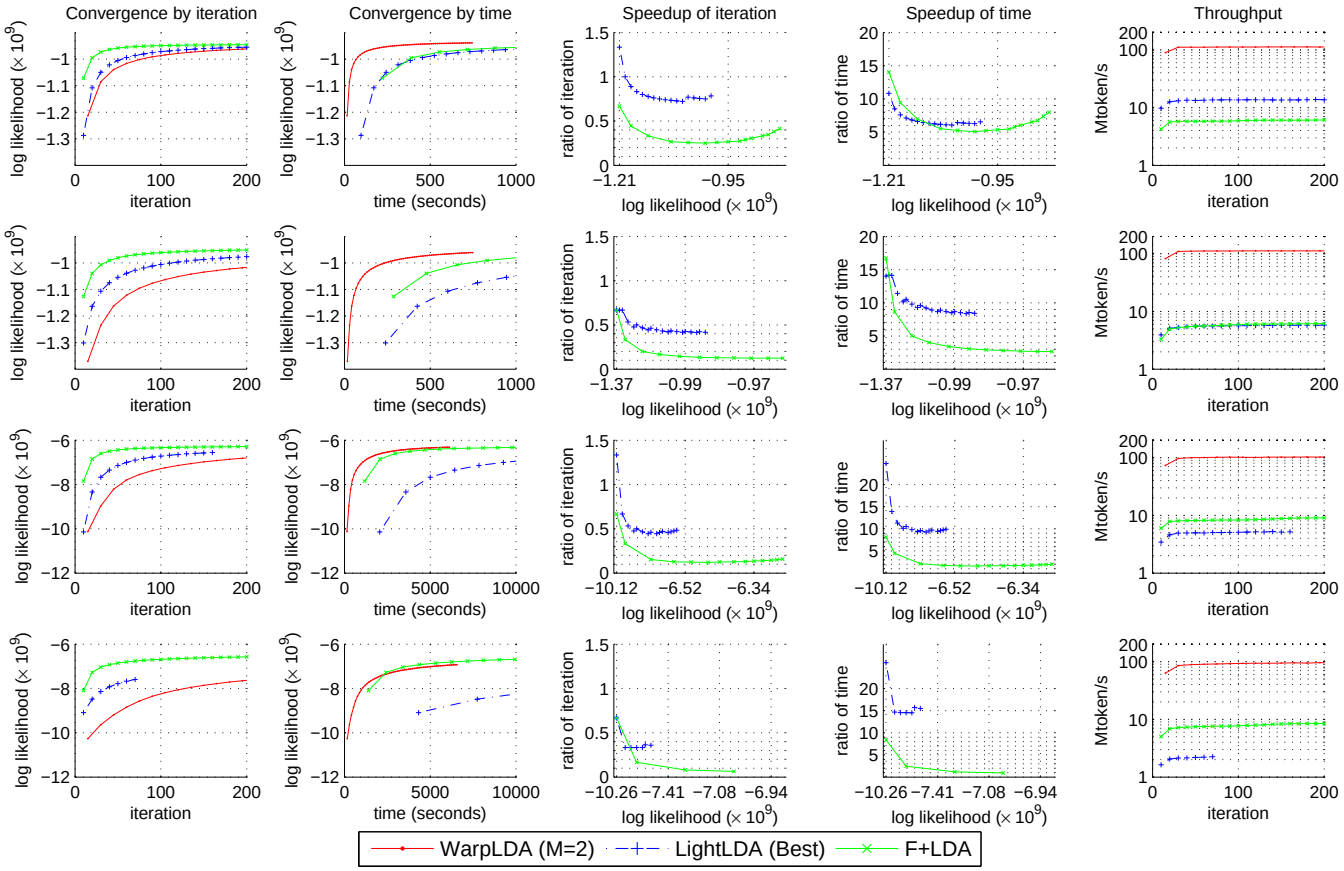


Figure 5: Convergence results on NYTimes (1st row: $K = 10^3$; 2nd row: $K = 10^4$) and PubMed (3rd row: $K = 10^4$; 4th row: $K = 10^5$). Each column corresponds to an evaluation metric (Please see text for details). The x-axis of column 3 and 4 are distorted for better resolution.

widely adopted log joint likelihood (log likelihood in short):

$$\mathcal{L} = \log p(\mathbf{W}, \mathbf{Z}|\boldsymbol{\alpha}, \beta) = \log \prod_d [\frac{\Gamma(\bar{\alpha})}{\Gamma(\bar{\alpha} + L_d)} \prod_k \frac{\Gamma(\alpha_k + C_{dk})}{\Gamma(\alpha_k)}] \prod_k [\frac{\Gamma(\bar{\beta})}{\Gamma(\bar{\beta} + C_k)} \prod_w \frac{\Gamma(\beta + C_{kw})}{\Gamma(\beta)}].$$

6.2 Speed of Convergence

We first analyze the convergence behaviors. Fig. 5 presents the single-machine results on the moderate-sized corpora, including NYTimes (first two rows) and PubMed (last two rows). We compare WarpLDA with a fixed $M = 2$ to both LightLDA and F+LDA. Each algorithm is run for a fixed number of iterations. LightLDA is sensitive with M because it affects the locality; We therefore pick up the M that leads to fastest convergence over time for LightLDA, which is $M = 4$ for the NYTimes dataset when $K = 10^3$, $M = 8$ for NYTimes when $K = 10^4$, $M = 8$ for PubMed when $K = 10^4$, and $M = 16$ for PubMed when $K = 10^5$. A larger M leads to longer running time per iteration, but the total number of iterations decreases in order to converge to a particular log-likelihood.

To have a full understanding, a diverse range of evaluation metrics are considered, including log-likelihood w.r.t the number of iterations (*1st column*), log-likelihood w.r.t running time (*2nd column*), the ratio of the iteration number of LightLDA (or F+LDA) over that of WarpLDA to get a particular log-likelihood (*3rd column*), the ratio of running time of LightLDA (or F+LDA) over that of WarpLDA to get a particular log-likelihood (*4th column*), and finally the throughput w.r.t the number of iterations (*5th column*). From the results, we have the following observations:

- WarpLDA converges to the same log-likelihood as other baselines (1st and 2nd columns), demonstrating the good quality;
- WarpLDA converges faster than F+LDA and LightLDA in terms of running time (2nd column), despite that it needs more iterations than the competitorsm to reach the same likelihood (1st column). Overall, WarpLDA is consistently 5-15x faster than LightLDA in all evaluations, and is faster than F+LDA when $K \leq 10^4$;
- Finally, WarpLDA is efficient — WarpLDA achieves 110M token/s throughput on these datasets with a *single machine*, much higher than that reported in previous works [1, 32, 31], which is typically less than 10M token/s.

Note that the convergence speed of F+LDA surpasses WarpLDA in later stages for the PubMed dataset when $K = 10^5$. This is mainly due to the difference between sparsity aware algorithms and MH-based algorithms. As K and D are increasing, the vector $\mathbf{c}_w$ becomes less concentrated so that MH-based algorithms require more samples from q^{word} to explore the state space, while the time complexity of the (exact) sparsity aware algorithms depends only on K_d which is upper bounded by L_d.

To show that WarpLDA is memory efficient, we compare the L3 cache miss rate of WarpLDA with that of LightLDA and F+LDA on the NYTimes and PubMed corpora. The cache miss rate is measured by PAPI. M is set to 1 for both WarpLDA and LightLDA. From Table 4, we can see that the L3 cache miss rate of WarpLDA

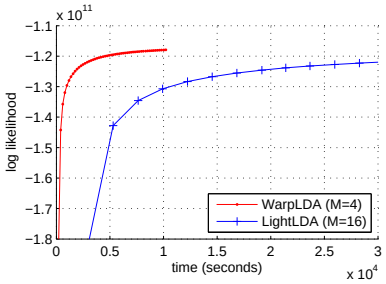


Figure 6: Convergence on ClueWeb12 (subset), $K = 10^4$.

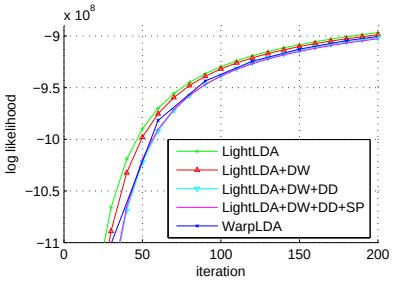


Figure 7: Quality of the MCEM solution of WarpLDA and the CGS solution of LightLDA. NYTimes, $K = 10^3$.

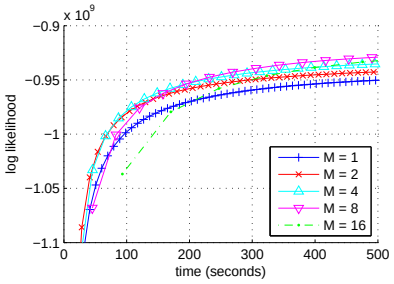


Figure 8: Impact of different M.

Table 4: L3 cache miss rate comparison, $M = 1$.

Setting	LightLDA	F+LDA	WarpLDA
NYTimes, $K = 10^3$	33%	77%	**17%**
NYTimes, $K = 10^4$	35%	53%	**13%**
PubMed, $K = 10^4$	38%	57%	**13%**
PubMed, $K = 10^5$	37%	17%	**5%**

is much lower than that of the competitors. This is reasonable because the size of randomly accessed memory per-document for WarpLDA fits in the L3 cache, while the competitors require to randomly access a large matrix with tens-of-gigabytes in size.

For the distributed setting, WarpLDA ($M = 4$) and LightLDA ($M = 16$) are compared on a 38-million-document ClueWeb12 (subset). Both algorithms are run on 32 machines. Fig. 6 presents the results. We can see that WarpLDA is about 10x faster than LightLDA to reach the same log-likelihood.

6.3 Quality of Solutions

Now we carefully analyze the quality of the solutions by WarpLDA, and show that the MCEM solution of WarpLDA is very similar with the CGS solution of LightLDA. Comparing the updates of LightLDA [32] and WarpLDA, we conclude that the differences that may affect the quality of the solutions are: (1) *delayed count update*: the counts $\mathbf{C}_k$ and $\mathbf{C}_d$ are updated instantly in LightLDA but delayed in WarpLDA, (2) q^{word}: For LightLDA $q^{\text{word}} \propto \frac{C_{wk}+\beta}{C_k+\beta}$ and for WarpLDA $q^{\text{word}} \propto C_{wk} + \beta$. Therefore, we vary these factors to gain insight on how each factor influences the convergence rate. We use the NYTimes corpus as an example and set $K = 10^3$ and $M = 1$. The algorithms in comparison are

- LightLDA: LightLDA with $M = 1$, in which $\mathbf{C}_d$ is updated instantly, and $\mathbf{C}_w$ is updated every 300 documents.
- LightLDA+DW: LightLDA, in which $\mathbf{C}_d$ is updated instantly, and $\mathbf{C}_w$ is updated per iteration.
- LightLDA+DW+DD: LightLDA, in which both $\mathbf{C}_d$ and $\mathbf{C}_w$ are updated per iteration.
- LightLDA+DW+DD+SP: LightLDA, in which both $\mathbf{C}_d$ and $\mathbf{C}_w$ are updated per iteration, using WarpLDA's q^{word}.
- WarpLDA: WarpLDA, in which both $\mathbf{C}_d$ and $\mathbf{C}_w$ are updated per iteration, using WarpLDA's q^{word}.

Fig. 7 shows the results, where all the algorithms require roughly the same number of iterations to converge to a particular log-likelihood, with the same M. This result shows that the delayed update and simple proposal distributions of WarpLDA do not affect the convergence much. Notice that the previous results in Fig. 5, where WarpLDA requires more iterations than LightLDA to converge to a particular log-likelihood, may look different from this result. This is because LightLDA uses a larger M in Fig. 5.

Our observation that delayed update does not affect convergence much seems to contradict with the previous observations for exact sampling algorithms [1]. A possible reason is that for MH-based algorithms (e.g., WarpLDA) the bottleneck of convergence is the efficiency of exploring $p(z_{dn})$ instead of the freshness of the counts; and thereby delayed update on the counts does not matter a lot.

We also analyze the impact of M on the solution quality in Fig. 8. We can see that as M gets larger WarpLDA converges faster. This is probably because of the bias induced by the finite-length MH chain. To keep the storage overhead small, we stick to a small M such as 1, 2 or 4, which leads to a sufficiently fast convergence.

6.4 Scalability Results

Finally, we demonstrate that WarpLDA can scale up to handle billion-scale documents on hundreds of machines.

Fig. 9(a) shows the multi-threading speedup result for WarpLDA with $M = 2$ and $K = 10^3$ on the NYTimes dataset. The throughput for a single core, a single CPU (12 cores), and 2 CPUs (24 cores) are 6M, 53M, and 104M tokens per second, respectively. The speedup of the 24-core version against the single-core version is 17x, which is good for such a memory intensive task. The 2-CPU (24 cores) version is faster than the single CPU (12 cores) version by 1.96x, indicating that our NUMA strategy is successful.

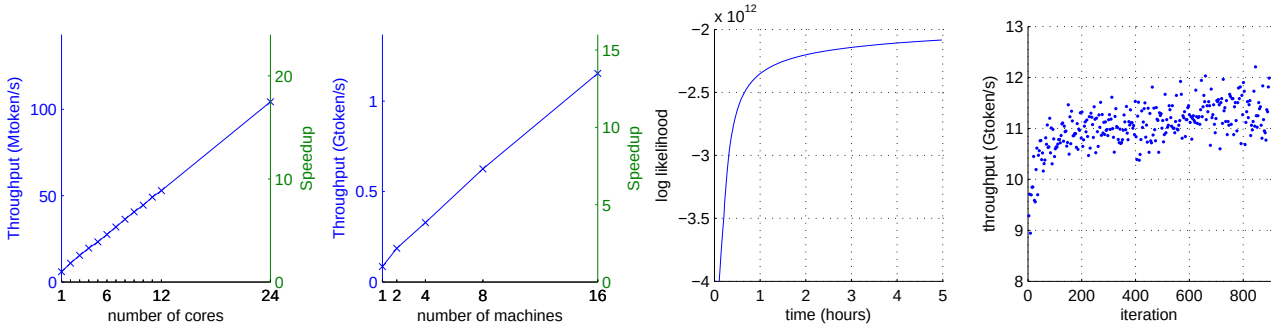
Fig. 9(b) shows the multi-machine speedup result for WarpLDA with $M = 1$ and $K = 10^4$ on the PubMed corpus. The throughput for 16 machines is 13.5x faster than the single machine version, demonstrating the good scalability.

To show our capacity of learning large-scale topic models, we learned $K = 10^6$ topics on the 639-million-document ClueWeb12 corpus, on 256 machines. The hyper-parameter β is set to 0.001 for finer grained topics, $M = 1$, and the number of iterations is 900. The convergence results are shown in Fig. 9(c). We can see that the run produces meaningful results in 5 hours.[6] Fig. 9(d) shows the throughput is an unprecedentedly 11G tokens/s with 256 machines.

7. CONCLUSIONS AND FUTURE WORK

We first analyze the memory efficiency of previous fast algorithms for LDA by the size of randomly accessed memory per-document, and conclude they are inefficient because of frequent

[6]The learned 1 million topics are available at `http://ml.cs.tsinghua.edu.cn/~jianfei/warplda.html`.



(a) Single machine (b) Distributed (c) Full ClueWeb12, convergence (d) Full ClueWeb12, throughput

Figure 9: Scalability results. a) multi-threading speedup on NYTimes, $K = 10^3$, $M = 4$; b) distributed speedup on PubMed, $K = 10^4$, $M = 1$; c, d) convergence and throughput on ClueWeb12, $K = 10^6$, $MH = 4$.

random accesses to large matrices. We then propose WarpLDA, an efficient algorithm for LDA which only requires to randomly access vectors that fit in the L3 cache, while maintaining the $O(1)$ time complexity. WarpLDA builds on an MCEM algorithm that enables delayed updates to decouple the accesses to $\mathbf{C}_d$ and $\mathbf{C}_w$, by some carefully designed ordering of visiting tokens. To implement WarpLDA in a memory efficient and scalable way, we design and implement a framework that supports manipulating the rows and columns of a distributed sparse matrix.

Extensive empirical studies in a wide range of testing conditions demonstrate that WarpLDA is consistently 5-15x faster than the state-of-the-art MH-based algorithms, and is faster than state-of-the-art sparsity-aware algorithms in most settings. WarpLDA achieves an unprecedentedly 11G token per second throughput which allows to train billion-scale corpora in hours.

In the future, we plan to combine WarpLDA with other promising directions of scaling up LDA and apply it to more sophisticated topic models to learn various types of topic structures.

Stochastic learning: Stochastic algorithms explore the statistical redundancy of a given corpus, and estimate the statistics (e.g., gradient) of the whole corpus by a random subset. When the estimation has low variance, faster convergence is expected. Examples include stochastic variational inference (SVI) [17], streaming variational Bayes [9], and stochastic gradient Riemann Langevin dynamics (SGRLD) [25]. These methods can be combined with fast sampling algorithms, e.g. WarpLDA, to create fast stochastic algorithms, e.g., Bhadury et al. combined SGRLD with LightLDA to scale up dynamic topic models [4].

GPU accelerations: There are some works on GPU acceleration for LDA [10, 34]. However, the algorithm they accelerate is $O(K)$. WarpLDA is a promising $O(1)$ option for GPU acceleration due to its single instruction multiple data (SIMD) nature. [7]

Non-conjugate topic models: Compared to the vanilla LDA, non-conjugate topic models capture richer types of statistical structures, e.g., correlation of topics [5, 12], temporal dynamics of topics [6, 4], or relationship to labels [38, 37]. These models typically have less sparsity structures to exploit, making sparsity-aware algorithms difficult to apply. We can still apply the MH-based WarpLDA to these models as a fast sampler for topic assignments.

8. ACKNOWLEDGMENTS

We thank Jinghui Yuan, Dong Yan and Arnab Bhadury for their help on this paper. The work was supported by the National Basic Research Program (973 Program) of China (Nos. 2013CB329403, 2012CB316301), National NSF of China (Nos. 61322308, 61332007), Tsinghua TNList Lab Big Data Initiative, and Tsinghua Initiative Scientific Research Program (No. 20141080934).

[7]Readers can verify by themselves that the tokens in one document / word can be processed simultaneously without data race.

9. REFERENCES

[1] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola. Scalable inference in latent variable models. In *WSDM*, 2012.

[2] A. Asuncion and D. Newman. Uci machine learning repository, 2007.

[3] A. Asuncion, M. Welling, P. Smyth, and Y. W. Teh. On smoothing and inference for topic models. In *UAI*, 2009.

[4] A. Bhadury, J. Chen, J. Zhu, and S. Liu. Scaling up dynamic topic models. In *WWW*, 2016.

[5] D. Blei and J. Lafferty. Correlated topic models. In *NIPS*, 2006.

[6] D. M. Blei and J. D. Lafferty. Dynamic topic models. In *ICML*, 2006.

[7] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *JMLR*, 3:993–1022, 2003.

[8] J. L. Boyd-Graber, D. M. Blei, and X. Zhu. A topic model for word sense disambiguation. In *EMNLP-CoNLL*, 2007.

[9] T. Broderick, N. Boyd, A. Wibisono, A. C. Wilson, and M. I. Jordan. Streaming variational bayes. In *NIPS*, 2013.

[10] J. Canny and H. Zhao. Bidmach: Large-scale learning with zero memory allocation. In *BigLearn Workshop, NIPS*, 2013.

[11] J. Chang and D. Blei. Relational topic models for document networks. In *AISTATS*, 2009.

[12] J. Chen, J. Zhu, Z. Wang, X. Zheng, and B. Zhang. Scalable inference for logistic-normal topic models. In *NIPS*, 2013.

[13] N. Chen, J. Zhu, F. Xia, and B. Zhang. Discriminative relational topic models. *IEEE Trans. on PAMI*, 37(5):973–986, 2015.

[14] W.-Y. Chen, J.-C. Chu, J. Luan, H. Bai, Y. Wang, and E. Y. Chang. Collaborative filtering for orkut communities: discovery of user latent behavior. In *WWW*, 2009.

[15] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the royal statistical society. Series B (methodological)*, pages 1–38, 1977.

[16] T. L. Griffiths and M. Steyvers. Finding scientific topics. *Proceedings of the National Academy of Sciences*, 101(suppl 1):5228–5235, 2004.

[17] M. D. Hoffman, D. M. Blei, C. Wang, and J. Paisley. Stochastic variational inference. *JMLR*, 14(1):1303–1347, 2013.

[18] T. Iwata, T. Yamada, and N. Ueda. Probabilistic latent semantic visualization: topic model for visualizing documents. In *KDD*, 2008.

[19] Z. G. Kingsley. Selective studies and the principle of relative frequency in language, 1932.

[20] A. Q. Li, A. Ahmed, S. Ravi, and A. J. Smola. Reducing the sampling complexity of topic models. In *KDD*, 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. In *OSDI*, 2014.

[22] S. Liu, X. Wang, J. Chen, J. Zhu, and B. Guo. Topicpanorama: a full picture of relevant topics. In *Proceedings of IEEE VAST*, 2014.

[23] T. Minka and J. Lafferty. Expectation-propagation for the generative aspect model. In *UAI*, 2002.

[24] D. Newman, P. Smyth, M. Welling, and A. U. Asuncion. Distributed inference for latent dirichlet allocation. In *NIPS*, 2007.

[25] S. Patterson and Y. W. Teh. Stochastic gradient riemannian langevin dynamics on the probability simplex. In *NIPS*, 2013.

[26] Y. W. Teh, D. Newman, and M. Welling. A collapsed variational bayesian inference algorithm for latent dirichlet allocation. In *NIPS*, 2006.

[27] A. J. Walker. An efficient method for generating discrete random variables with general distributions. *ACM Transactions on Mathematical Software (TOMS)*, 3(3):253–256, 1977.

[28] Y. Wang, X. Zhao, Z. Sun, H. Yan, L. Wang, Z. Jin, L. Wang, Y. Gao, J. Zeng, Q. Yang, et al. Towards topic modeling for big data. *ACM Transactions on Intelligent Systems and Technology*, 2014.

[29] X. Wei and W. B. Croft. Lda-based document models for ad-hoc retrieval. In *SIGIR*. ACM, 2006.

[30] L. Yao, D. Mimno, and A. McCallum. Efficient methods for topic model inference on streaming document collections. In *KDD*, 2009.

[31] H.-F. Yu, C.-J. Hsieh, H. Yun, S. Vishwanathan, and I. S. Dhillon. A scalable asynchronous distributed algorithm for topic modeling. In *WWW*, 2015.

[32] J. Yuan, F. Gao, Q. Ho, W. Dai, J. Wei, X. Zheng, E. P. Xing, T.-Y. Liu, and W.-Y. Ma. Lightlda: Big topic models on modest compute clusters. In *WWW*, 2015.

[33] M. Zaheer, M. Wick, J.-B. Tristan, A. Smola, and G. L. Steele Jr. Exponential stochastic cellular automata for massively parallel inference. In *AISTATS*, 2015.

[34] H. Zhao, B. Jiang, and J. Canny. Same but different: Fast and high-quality gibbs parameter estimation. *arXiv:1409.5402*, 2014.

[35] X. Zheng, Y. Yu, and E. P. Xing. Linear time samplers for supervised topic models using compositional proposals. In *KDD*, 2015.

[36] J. Zhu, A. Ahmed, and E. P. Xing. MedLDA: Maximum margin supervised topic models. *JMLR*, 13:2237–2278, 2012.

[37] J. Zhu, X. Zheng, L. Zhou, and B. Zhang. Bayesian logistic supervised topic models with data augmentation. In *ACL*, 2013.

[38] J. Zhu, X. Zheng, L. Zhou, and B. Zhang. Scalable inference in max-margin topic models. In *KDD*, 2013.

APPENDIX

A. FULL CODE OF WARPLDA

Alg. 2 is the full pseudo-code of WarpLDA, with random positioning for sampling the document proposal and alias sampling for sampling the word proposal.

Algorithm 2 The WarpLDA algorithm, where $\text{Dice}(K)$ draws a sample from $\{0, \dots, K-1\}$. Alias table is used for sampling from $\mathbf{C}_w$ and positioning is used for sampling from $\mathbf{C}_d$.

// Initialize
$z_{dn} \leftarrow \text{Dice}(K), \forall d, n$
for $iteration \leftarrow 1$ **to** I **do**
  // Word phase: $C_{wk}, \pi^{\text{doc}}, q^{\text{word}}$
  for $w \leftarrow 1$ **to** V **do**
    // Compute C_{wk} on the fly
    $C_{wk} \leftarrow \sum_{n=1}^{L_w} \mathbb{I}(z_{wn} = k), k = 1, \dots, K$
    // Simulate q^{doc} chain with samples from last iteration
    for $n \leftarrow 1$ **to** L_w **do**
      for $i \leftarrow 1$ **to** MH **do**
        $s \leftarrow z_{wn}^{(i-1)}, t \leftarrow z_{wn}^{(i)}$
        $\pi \leftarrow \min\{1, \frac{C_{wt}+\beta}{C_{ws}+\beta} \frac{C_s+\bar{\beta}V}{C_t+\bar{\beta}}\}$
        $z_{wn} \leftarrow t$ with probability π
      end for
    end for
    // Update C_{wk}
    $C_{wk} \leftarrow \sum_{n=1}^{L_w} \mathbb{I}(z_{wn} = k), k = 1, \dots, K$
    $urn \leftarrow \text{BuildAlias}(\mathbf{C}_w)$
    // Draw samples from q^{word}
    for $n \leftarrow 1$ **to** L_w **do**
      for $i \leftarrow 1$ **to** MH **do**
        $z_{wn}^{(i)} \leftarrow urn.\text{Draw}()$
      end for
    end for
  end for
  // Document phase: $C_{dk}, \pi^{\text{word}}, q^{\text{doc}}$
  for $d \leftarrow 1$ **to** D **do**
    // Compute C_{dk} on the fly
    $C_{dk} \leftarrow \sum_{n=1}^{L_d} \mathbb{I}(z_{dn} = k), k = 1, \dots, K$
    // Simulate q^{word} chain with samples from last iteration
    for $n \leftarrow 1$ **to** L_d **do**
      for $i \leftarrow 1$ **to** MH **do**
        $s \leftarrow z_{dn}^{(i-1)}, t \leftarrow z_{dn}^{(i)}$
        $\pi \leftarrow \min\{1, \frac{C_{dt}+\alpha_t}{C_{ds}+\alpha_s} \frac{C_s+\bar{\beta}}{C_t+\bar{\beta}}\}$
        $z_{dn} \leftarrow t$ with probability π
      end for
    end for
    // Draw samples from q^{doc}
    for $n \leftarrow 1$ **to** L_d **do**
      for $i \leftarrow 1$ **to** MH **do**
        $z_{dn}^{(i)} \leftarrow \begin{cases} z_{d,\text{Dice}(L_d)} & \text{with probability } \frac{L_d}{L_d+\bar{\alpha}} \\ \text{Dice}(K) & \text{otherwise} \end{cases}$
      end for
    end for
  end for
end for