PLDA+: Parallel Latent Dirichlet Allocation with Data Placement and Pipeline Processing

ZHIYUAN LIU, YUZHOU ZHANG, EDWARD Y. CHANG
Google Inc.
MAOSONG SUN
Tsinghua University

Previous methods of distributed Gibbs sampling for LDA run into either memory or communication bottleneck. To improve scalability, we propose four strategies: *data placement*, *pipeline processing*, *word bundling*, and *priority-based scheduling*. Experiments show that our strategies significantly reduce the unparallelizable communication bottleneck and achieve good load balancing, and hence improve scalability of LDA.

Categories and Subject Descriptors: G.3 [**Probability and Statistics**]: Probabilistic Algorithms; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval— *Clustering*; I.2.7 [**Artificial Intelligence**]: Natural Language Processing—*Text Analysis*

General Terms: Algorithms

Additional Key Words and Phrases: Topic Models, Gibbs Sampling, Latent Dirichlet Allocation, Distributed Parallel Computations

1. INTRODUCTION

Latent Dirichlet Allocation (LDA) was first proposed by Blei, Ng and Jordan to model documents [Blei et al. 2003]. Each document is modeled as a mixture of K latent topics, where each topic, k, is a multinomial distribution $\boldsymbol{\phi}_k$ over a W-word vocabulary. For any document d_j, its topic mixture $\boldsymbol{\theta}_j$ is a probability distribution drawn from a Dirichlet prior with parameter α. For each i^{th} word x_{ij} in d_j, a topic $z_{ij} = k$ is drawn from $\boldsymbol{\theta}_j$, and x_{ij} is drawn from $\boldsymbol{\phi}_k$. The generative process for LDA is thus given by

$$\boldsymbol{\theta}_j \sim Dir(\alpha), \boldsymbol{\phi}_k \sim Dir(\beta), z_{ij} = k \sim \boldsymbol{\theta}_j, x_{ij} \sim \boldsymbol{\phi}_k, \tag{1}$$

where $Dir(*)$ denotes the Dirichlet distribution. The graphical model for LDA is illustrated in Fig. 1, where the observed variables, i.e., words x_{ij} and hyper parameters α and β, are shaded.

Using Gibbs sampling to learn LDA, the computation complexity is K multiplied by

Authors' address: Zhiyuan Liu, Yuzhou Zhang and Edward Y. Chang, Google China, e-mail: lzy.thu@gmail.com, yuzhou.zh@gmail.com, edchang@google.com. Maosong Sun, Department of Computer Science and Technology, Tsinghua University, e-mail: sms@tsinghua.edu.cn. Zhiyuan Liu and Yuzhou Zhang are Ph.D. students in the Department of Computer Science and Technology, Tsinghua University. This work was done during Liu and Zhang's research internships in Google China.

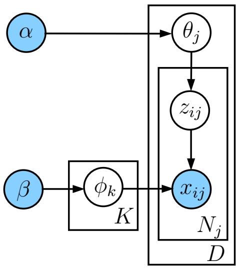

Fig. 1: The graphical model for LDA.

the total number of word occurrences in the training corpus. Prior work has explored two main parallelization approaches for speeding up LDA: 1) parallelizing on loosely-coupled distributed computers, and 2) parallelizing on tightly-coupled multi-core CPUs or GPUs (Graphics Processing Units). Representative loosely-coupled distributed algorithms are Dirichlet Compound Multinomial LDA (DCM-LDA) [Mimno and McCallum 2007], Approximate Distributed LDA (AD-LDA) [Newman et al. 2007], and Asynchronous Distributed LDA (AS-LDA) [Asuncion et al. 2008], which perform Gibbs sampling on computers that do not share memory. This distributed approach may suffer from high inter-computer communication cost, which limits achievable speedup. The tightly-coupled approach uses multi-core CPUs or GPUs with shared memory (e.g., the work of Yan, et al. [2009]). Such a shared-memory approach reduces inter-process communication time. However, once the processors and memory have been configured, the architecture is inflexible when faced with changes in computation demands, and the need to schedule simultaneous tasks with mixed resource requirements. (We discuss related work in greater detail in Section 2.)

In this work, we improve the scalability of the distributed approach by reducing inter-computer communication time. Our algorithm, which we name PLDA+, employs four inter-dependent strategies:

(1) *Data placement*. Data placement aims to separate CPU-bound tasks and communication-bound tasks onto two sets of processors. Data placement enables us to employ a pipeline scheme (discussed next), to mask communication by computation.

(2) *Pipeline processing*. To ensure that a CPU-bound processor is not blocked by communication, PLDA+ conducts Gibbs sampling for a *word bundle* while performing inter-computer communication on the background. Suppose Gibbs sampling is performed on the words 'foo' and 'bar'. PLDA+ fetches the metadata for the word 'bar' while performing Gibbs sampling on the word 'foo'. The communication time for fetching the metadata of 'bar' is masked by the computation time for sampling 'foo'.

(3) *Word bundling*. In order to ensure that communication time can be effectively masked, the CPU time must be long enough. Revisiting the example of sampling 'foo' and 'bar', the CPU time for sampling the word 'foo' should be longer than the communication time for the word 'bar' in order to mask the communication time. Suppose we performed Gibbs sampling according to the order of words in documents, each Gibbs sampling time unit would be too short to mask the required communication time. Since LDA treats a document as a bag of words and entirely ignores word order, we can flexibly process words on a processor in any order without considering

Table I: Symbols associated with LDA used in this paper.

D	Number of documents.
K	Number of topics.
W	Vocabulary size.
N	Number of words in the corpus.
x_{ij}	The i^{th} word in d_j document.
z_{ij}	Topic assignment for word x_{ij}.
C_{kj}	Number of topic k assigned to d_j document.
C_{wk}	Number of word w assigned to topic k.
C_k	Number of topic k in corpus.
C^{doc}	Document-topic count matrix.
C^{word}	Word-topic count matrix.
C^{topic}	Topic count matrix.
$\boldsymbol{\theta}_j$	Probability of topics given document d_j.
$\boldsymbol{\phi}_k$	Probability of words given topic k.
α	Dirichlet prior.
β	Dirichlet prior.
P	Number of processors.
$\|P_w\|$	Number of P_w processors.
$\|P_d\|$	Number of P_d processors.
p_i	The i^{th} processor.

document boundaries. Word bundling combines words into large computation units.

(4) *Priority-based scheduling*. *Data placement* and *word bundling* are static allocation strategies for improving pipeline performance. However, run time factors would almost always affect the effectiveness of a static allocation scheme. Therefore, PLDA+ employs a priority-based scheduling scheme to smooth out run-time bottlenecks.

The above four strategies must work together to improve speedup. For instance, without word bundling, pipeline processing is futile because of short computation units. Without distributing the metadata of word bundles, communication bottlenecks at the *master* processor could cap scalability. By lengthening the computation units via word bundling, while shortening communication units via data placement, we can achieve more effective pipeline processing. Finally, a priority-based scheduler helps smooth out unexpected runtime imbalances in workload.

The rest of the paper is organized as follows: We first present LDA and related distributed algorithms in Section 2. In Section 2.3 we present PLDA, an MPI implementation of Approximate Distributed LDA (AD-LDA). In Section 3 we analyze the bottleneck of PLDA and depict PLDA+. Section 4 demonstrates that the speedup of PLDA+ on large-scale document collections significantly outperforms PLDA. Section 5 offers our concluding remarks. For the convenience of readers, we summarize the notations used in this paper in Table I.

2. LDA OVERVIEW

Similar to most previous work [Griffiths and Steyvers 2004], we use symmetric Dirichlet priors in LDA for simplicity. Given the observed words $\boldsymbol{x}$, the task of inference for LDA is to compute the posterior distribution of the latent topic assignments $\boldsymbol{z}$, the topic mixtures of documents $\boldsymbol{\theta}$, and the topics $\boldsymbol{\phi}$.

2.1 LDA Learning

Griffiths and Steyvers [2004] proposed using Gibbs sampling, a Markov-chain Monte Carlo (MCMC) method, to perform inference for LDA. By assuming a Dirichlet prior β on $\boldsymbol{\phi}$, $\boldsymbol{\phi}$ can be integrated (hence removed from the equation) using the Dirichlet-multinomial conjugacy. MCMC is widely used as an inference method for latent topic models, e.g., Author-Topic Model [Rosen-Zvi et al. 2010], Pachinko Allocation [Li and McCallum 2006], and Special Words with Background Model [Chemudugunta et al. 2007]. Moreover, since the memory requirement of VEM is not nearly as scalable as that of MCMC [Newman et al. 2009], most existing distributed methods for LDA use Gibbs sampling for inference, e.g., DCM-LDA, AD-LDA, and AS-LDA. In this paper we focus on Gibbs sampling for approximate inference. In Gibbs sampling, it is usual to integrate out the mixtures $\boldsymbol{\theta}$ and topics $\boldsymbol{\phi}$ and just sample the latent variables $\boldsymbol{z}$. The process is called *collapsing*. When performing Gibbs sampling for LDA, we maintain two matrices: a word-topic count matrix C^{word} in which each element C_{wk} is the number of word w assigned to topic k, and a document-topic count matrix C^{doc} in which each element C_{kj} is the number of topic k assigned to document d_j. Moreover, we maintain a topic count vector C^{topic} in which each element C_k is the number of topic k assignments in document collection. Given the current state of all but one variable z_{ij}, the conditional probability of z_{ij} is

$$p(z_{ij} = k|\boldsymbol{z}^{\neg ij}, \boldsymbol{x}^{\neg ij}, x_{ij} = w, \alpha, \beta) \propto \frac{C_{wk}^{\neg ij} + \beta}{C_k^{\neg ij} + W\beta} \left(C_{kj}^{\neg ij} + \alpha \right), \tag{2}$$

where $\neg ij$ means that the corresponding word is excluded in the counts. Whenever z_{ij} is assigned with a new topic drawn from Eq. (2), C^{word}, C^{doc} and C^{topic} are updated. After enough sampling iterations to burn in the Markov chain, $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$ are estimated.

2.2 LDA Performance Enhancement

Various approaches have been explored for speeding up LDA. Relevant parallel methods for LDA include:

- Mimno and McCallum [2007] proposed Dirichlet Compound Multinomial LDA (DCM-LDA), where the datasets are distributed to processors, Gibbs sampling is performed on each processor independently without any communication between processors, and finally a global clustering of the topics is performed.
- Newman, et al. [2007] proposed Approximate Distributed LDA (AD-LDA), where each processor performs a local Gibbs sampling iteration followed by a global update using a reduce-scatter operation. Since the Gibbs sampling on each processor is performed with the local word-topic matrix, which is only updated at the end of each iteration, this method is called *approximate* distributed LDA.
- In [Asuncion et al. 2008], a purely asynchronous distributed LDA was proposed, where no global synchronization step like in [Newman et al. 2007] is required. Each processor performs a local Gibbs sampling step followed by a step of communicating with other *random* processors. In this paper we label this method as AS-LDA.
- Yan, et al. [2009] proposed parallel algorithms of Gibbs sampling and VEM for LDA on GPUs. A GPU has massively built-in parallel processors with shared memory.

Besides these parallelization techniques, the following optimizations can reduce LDA model learning computation cost:

—Gomes, et al. [2008] presented an enhancement of the VEM algorithm using a bounded amount of memory.

—Porteous, et al. [2008] proposed a method to accelerate the computation of Eq. (2). The acceleration is achieved by no approximations but using the property that the topic probability vectors for document d_j, $\boldsymbol{\theta}_j$, are sparse in most cases.

2.3 PLDA: An MPI Implementation of AD-LDA

We previously implemented PLDA [Wang et al. 2009], an MPI implementation of AD-LDA [Newman et al. 2007]. PLDA has been successfully applied in real-world applications such as communication recommendation [Chen et al. 2009]. AD-LDA distributes D training documents over P processors, with $D_p = D/P$ documents on each processor. AD-LDA partitions document content $\boldsymbol{x} = \{\boldsymbol{x}_d\}_{d=1}^{D}$ into $\{\boldsymbol{x}_{|1}, \ldots, \boldsymbol{x}_{|P}\}$ and the corresponding topic assignments $\boldsymbol{z} = \{\boldsymbol{z}_d\}_{d=1}^{D}$ into $\{\boldsymbol{z}_{|1}, \ldots, \boldsymbol{z}_{|P}\}$, where $\boldsymbol{x}_{|p}$ and $\boldsymbol{z}_{|p}$ exist only on processor p. The document-topic count matrix, C^{doc}, is likewise distributed and we represent the processor-specific document-topic count matrices as $C^{doc}_{|p}$. Each processor maintains its own copy of the word-topic count matrix, C^{word}. Moreover, we use $C^{word}_{|p}$ to temporarily store word-topic counts accumulated from local documents' topic assignments on each processor. In each Gibbs sampling iteration, each processor p updates $\boldsymbol{z}_{|p}$ by sampling every $z_{ij|p} \in \boldsymbol{z}_{|p}$ from the approximate posterior distribution:

$$p(z_{ij|p} = k \mid \boldsymbol{z}^{\neg ij}, \boldsymbol{x}^{\neg ij}, x_{ij|p} = w) \propto \frac{C^{\neg ij}_{wk} + \beta}{C^{\neg ij}_{k} + W\beta} \left(C^{\neg ij}_{kj|p} + \alpha \right), \tag{3}$$

and updates $C^{doc}_{|p}$ and $C^{word}_{|p}$ according to the new topic assignments. After each iteration, each processor recomputes word-topic counts for its local documents $C^{word}_{|p}$ and uses an AllReduce operation to reduce and broadcast the new C^{word} to all processors. One can refer to [Wang et al. 2009] for the MPI implementation details of AD-LDA.

We have also implemented AD-LDA on MapReduce [Dean and Ghemawat 2004; Chu et al. 2006] as reported in [Wang et al. 2009]. Using MapReduce, many operations can be carried out by combining three basic phases: mapping, shuffling and reducing. We used MapReduce to implement *AllReduce*. However, before and after each iteration of the MapReduce-based AD-LDA, a disk IO is required to fetch and update the word-topic matrix at the *master* processor. In addition, local data must also be written onto disks. The benefit of forcing IOs between iterations is tolerating faults. However, using MPI, a fault recovery scheme can be more efficiently implemented via lazy IOs after the completion of each iteration. The primary reason for conducing IOs is because MapReduce cannot ensure two consecutive iterations of sampling the same set of data being scheduled on the same processor. Thus, documents and metadata (document-topic counts) must be fetched into memory at the beginning of each iteration even in the absence of a fault. Certainly, these shortcomings of MapReduce can be improved. But MPI seemed to be a more attractive choice at the time when this research was conducted.

3. PLDA+: AN ENHANCED DISTRIBUTED LDA

To further speed up LDA, the PLDA+ algorithm employs four inter-dependent strategies to reduce inter-computer communication cost: data placement, pipeline processing, word bundling, and priority-based scheduling.

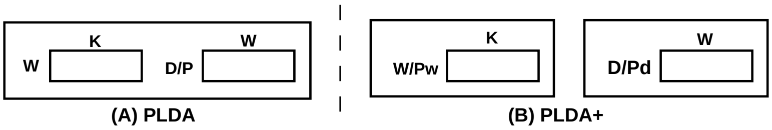

Fig. 2: The assignments of documents and word-topic count matrix for PLDA and PLDA+.

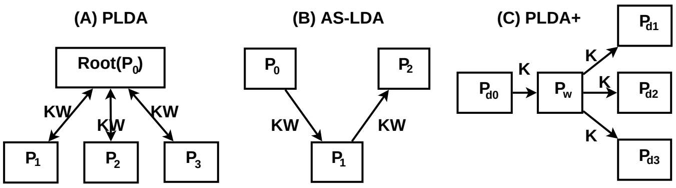

Fig. 3: The spread patterns of the updated topic distribution of a word from one processor for PLDA, AS-LDA and PLDA+.

3.1 Bottlenecks for PLDA

As presented in the previous section, in PLDA, D documents are distributed over P processors with approximately D/P documents on each processor. This is shown with a D/P-by-W matrix in Fig. 2(A), where W indicates the vocabulary of document collection. The word-topic count matrix is also distributed, with each processor keeping a local copy, which is the W-by-K matrix in Fig. 2(A).

In PLDA, after each iteration of Gibbs sampling, local word-topic counts on each processor are globally synchronized. This synchronization process is expensive partly because a large amount of data is sent and partly because the synchronization starts only when the slowest processor has completed its work. To avoid unnecessary delays, AS-LDA [Asuncion et al. 2008] does not perform global synchronization like PLDA. In AS-LDA a processor only synchronizes word-topic counts with another finished processor. However, since word-topic counts can be outdated, the sampling process can take a larger number of iterations than that PLDA does to converge. Fig. 3(A) and Fig. 3(B) illustrate the spread patterns of the updated topic distribution for a word from one processor to the others for PLDA and AS-LDA. PLDA has to synchronize all word updates after a full Gibbs sampling iteration, whereas AS-LDA performs updates only with a small subset of processors. The memory requirements for both PLDA and AS-LDA are O(KW), since the whole word-topic matrix is maintained on all processors.

Although they apply different strategies for model combination, existing distributed methods share two characteristics:

—The methods have to maintain all word-topic counts in memory for each processor.

—The methods have to send and receive the whole word-topic matrix between processors for updates.

For the former characteristic, suppose we want to estimate a ϕ with W words and K topics from a large-scale dataset. When either W or K is large to a certain extent, the memory requirement will exceed that available on a typical processor. For the latter characteristic, the communication bottleneck caps the potential for speeding up the algorithm. A study of high performance computing [Graham et al. 2005] shows that floating-point instructions historically improve at 59% per year, but inter-processor bandwidth improves 26% per year, and inter-processor latency reduces only 15% per year. The communication bottleneck will only exacerbate over additional years.

3.2 Strategies of PLDA+

Let us first introduce pipeline-based Gibbs sampling. The pipeline technique has been used in many applications to enhance throughput, such as the instruction pipeline in modern CPUs [Shen and Lipasti 2005] and in graphics processors [Blinn 1991]. Although pipeline does not decrease the time for a job to be processed, it can efficiently improve throughput by overlapping communication with computation. Fig. 4 illustrates the pipeline-based Gibbs sampling for four words, w_1, w_2, w_3 and w_4. Fig. 4(A) demonstrates the case when $t_s \geq t_f + t_u$, and Fig. 4(B) the case when $t_s < t_f + t_u$, where t_s, t_f and t_u denote the time for Gibbs sampling, fetching the topic distribution, and updating the topic distribution, respectively.

In Fig. 4(A), PLDA+ begins by fetching the topic distribution for w_1. Then it begins Gibbs sampling on w_1, and at the same time, it fetches the topic distribution for w_2. After it has finished Gibbs sampling for w_1, PLDA+ updates the topic distribution for w_1 on P_w processors. When $t_s \geq t_f + t_u$, PLDA+ can begin Gibbs sampling on w_2 immediately after it has completed sampling for w_1. The total ideal time for PLDA+ to process W words will be $Wt_s + t_f + t_u$. Fig. 4(B) shows a suboptimal scenario where the communication time cannot be entirely masked. PLDA+ is not able to begin Gibbs sampling for w_3 until w_2 has been updated and w_3 fetched. The example shows that in order to successfully mask communication, we must schedule tasks to ensure as much as possible that $t_s \geq t_f + t_u$.

To make the pipeline strategy effective or $t_s \geq t_f + t_u$, PLDA+ divides processors into two types: one maintains documents and the document-topic count matrix to perform Gibbs sampling (P_d processors), while the other stores and maintains the word-topic count matrix (P_w processors). The structure is shown in Fig. 2(B). During each iteration of Gibbs sampling, a P_d processor assigns a new topic to a word in a typical three-stage process:

(1) Fetch the word's topic distribution from a P_w processor.

(2) Perform Gibbs sampling and assign a new topic to the word.

(3) Update the P_w processors maintaining the word.

The corresponding spread pattern for PLDA+ is illustrated in Fig. 3(C), which avoids both the global synchronization of PLDA and the large number of iterations required by AS-LDA for convergence.

One key property that PLDA+ takes advantage of is that each round of Gibbs sampling can be performed in any word order. Since LDA models a document as a bag of words and ignores word order, we can perform Gibbs sampling according to any word order as if we reordered words in bags. When a word that occurs multiple times in the documents of a P_d processor, all instances of that word can be processed together. Moreover, for words that occur infrequently, we bundle them with words that occur more frequently to ensure that t_s is sufficiently long. In fact, if we know $t_f + t_u$, we can decide how many word-occurrences to process in each Gibbs sampling batch to ensure that $t_s - (t_f + t_u)$ is minimized.

To perform Gibbs sampling word by word, PLDA+ builds word indexes to documents on each P_d processor. We then organize words in a *circular queue* as shown in Fig. 5.

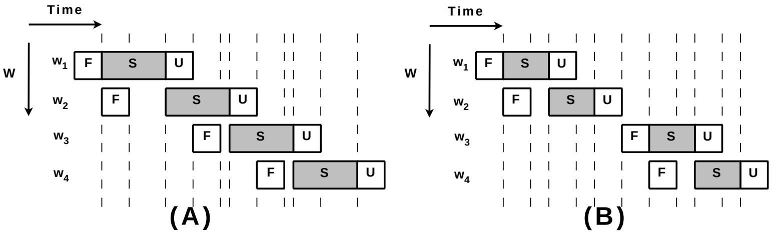

Fig. 4: Pipeline-based Gibbs sampling in PLDA+. (A): $t_s \geq t_f + t_u$. (B): $t_s < t_f + t_u$. In this figure, *F* indicates the fetching operation, *U* indicates the updating operation, and *S* indicates the Gibbs sampling operation.

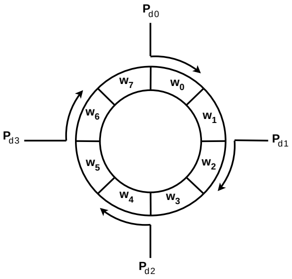

Fig. 5: Vocabulary circular queue in PLDA+.

Gibbs sampling is performed by going around the circular queue. To avoid concurrent access to the same words, we schedule different processors to begin at different positions of the queue. For example, Fig. 5 shows four P_d processors, P_{d0}, P_{d1}, P_{d2} and P_{d3} start their first word from w_0, w_2, w_4 and w_6, respectively. To ensure that this scheduling algorithm works, PLDA+ must also distribute the word-topic matrix in a circular fashion on P_w processors. This static allocation scheme enjoys two benefits. First, the workload among P_w processors can be relatively balanced. Second, avoiding two P_d nodes from concurrently updating the same word can roughly maintain serializability of the word-topic matrix on P_w nodes. Please note that the distributed scheme of PLDA+ ensures stronger serializability than PLDA because a P_d node of PLDA+ can obtain the word-topic matrix updates of other P_d nodes in the same Gibbs sampling iteration. The detailed description of word placement are presented in Section 3.3.1.

Although word placement can be performed in an optimal way, scheduling must deal with run-time dynamics. First, some processors may run faster than others, and this may build up bottlenecks at some of the P_w processors. Second, when multiple requests are pending, the scheduler must be able to set priorities based on request deadlines. The details of PLDA+'s priority-based scheduling scheme are described in Section 3.4.3.

3.3 Algorithm for P_w Processors

The task of the P_w processors is to process, fetch and update queries from P_d processors. PLDA+ distributes the word-topic matrix to P_w processors according to the words contained in the matrix. After placement, each P_w processor keeps approximately $W/|P_w|$

words with their topic distributions.

3.3.1 *Word Placement over P_w Processors.* The goal of word placement is to ensure *spatial* load balancing. We would like to make sure that all processors receive about the same number of requests in a round of Gibbs sampling.

For bookkeeping, we maintain two data structures. First, we use m_i to record the number of P_d processors on which a word w_i resides, which is also the weight of the word. For W words, we maintain a vector $\vec{m} = (m_1, \ldots, m_W)$. The second data structure keeps track of each P_w processor's workload, or the sum of weights of all words on that processor. The workload vector is denoted as $\vec{l} = (l_1, \ldots, l_{|P_w|})$.

A simple placement method is to place words independently and uniformly at random on P_w processors. This method is referred to as *Random Word Placement*. Unfortunately, this placement method may cause frequent load imbalances. To balance workload, we use the *Weighted Round-Robin* method for word placement. We first sort words in *descending* order by their weights. We then pick the word with the largest weight from the vocabulary (e.g., w_i), place it on the P_w processor (e.g., pw) with the least workload, and then update the workload of pw. This placement process is repeated until all words have been placed. Weighted Round-Robin has been empirically shown to achieve balanced load with high probability [Berenbrink et al. 2008].

3.3.2 *Processing Requests from P_d Processors.* After placing words with their topic distributions on P_w processors, the P_w processors begin to process requests from the P_d processors. A P_w processor pw first builds its associated word-topic count matrix $C^{word}_{|pw}$ by receiving initial word-topic counts from all P_d processors. Then the P_w processor pw begins to process requests from P_d processors. In PLDA+ we define three types of requests:

—$fetch(w_i, pw, pd)$: a request for fetching the topic distribution of a word w by a P_d processor pd. For each request, the P_w processor pw returns the topic distribution $C^{word}_{w|pw}$ of the word w, which will be used as $C^{\neg ij}_{wk}$ in Eq. (2) for Gibbs sampling.

—$update(w, \vec{u}, pw)$: a request for updating the topic distribution of a word w using the update information $\vec{u}$ on pd. The P_w processor updates the topic distribution of the word w using $\vec{u}$.

—$fetch(pw, pd)$: a request for fetching the overall topic counts on a P_w processor pw by a P_d processor pd. The P_w Processor pw sums up the topic distributions for all words on pw as a vector $C^{topic}_{|pw}$. Once all $C^{topic}_{|pw}$ are fetched from each P_w processor by pd, they are summed up and used as $C^{\neg ij}_{k}$ in Eq. (2) for Gibbs sampling.

Each P_w processor handles all requests related to the words it is responsible for maintaining. To ensure that requests are served in a timely manner, we employed a priority scheme sorted by request deadlines. According to its local word processing order, a P_d processor needs communication completion for its fetch requests at various time units. When the P_d processor sends its requests to P_w processors, deadlines are set in the request header. A P_w processor serves waiting requests based on their deadlines.

3.4 Algorithm for P_d Processors

The algorithm for P_d processors executes according to the following steps:

(1) At the beginning, it allocates documents over P_d processors and then builds an inverted index for documents on each P_d processor.

(2) It groups the words in the vocabulary into *bundles* for performing Gibbs sampling and sending requests.

(3) It schedules word bundles to minimize communication bottlenecks.

(4) Finally, it performs pipeline-based Gibbs sampling iteratively until the termination condition is met.

In the following, we present the four steps in detail.

3.4.1 *Document Allocation and Building an Inverted Index.* Before performing Gibbs sampling, we first have to distribute D documents to P_d processors. The goal of document allocation is to achieve good CPU load balance among P_d processors. PLDA may suffer from imbalanced load since it has a global synchronization phase at the end of each Gibbs sampling iteration, which may force fast processors to wait for the slowest processor. In contrast, Gibbs sampling in PLDA+ is performed with no synchronization requirement. In other words, a fast processor can start its next round of sampling without having to wait for a slow processor. However, we also do not want some processors to be substantially slow and miss too many cycles of Gibbs sampling. This will result in the similar shortcoming that AS-LDA suffers — taking a larger number of iterations to converge. Thus, we would like to allocate documents to processors in a balanced fashion. This is achieved by employing *Random Document Allocation*. Each P_d processor gets approximate $D/|P_d|$ documents. The time complexity of this allocation step is $O(D)$.

After documents have been distributed, we build an inverted index for the documents of each P_d processor. Using this inverted index, each time a P_d processor fetches the topic distribution of a word w, it performs Gibbs sampling for all instances of w on that processor. After sampling, the processor sends back the updated topic distribution to the corresponding P_w processor. The clear benefit is that for multiple occurrences of a word on a processor, we only need to perform two communications, one fetch and one update, substantially reducing communication cost. The index structure for each word w is:

$$w \rightarrow \{(d_1, z_1), (d_1, z_2), (d_2, z_1) \ldots\}, \tag{4}$$

in which, w occurs in document d_1 for 2 times and there are 2 entries. In implementation, to save memory, we will record all occurrences of w in d_1 as one entry, $(d_1, \{z_1, z_2\})$.

3.4.2 *Word bundle.* Bundling words is to prevent the duration of Gibbs samplings from being too short to mask communication. Use an extreme example: a word takes place only once on a processor. Performing Gibbs sampling on that word takes a much shorter time than the time required to fetch and update the topic distribution of that word. The remedy is intuitive: combining a few words into a bundle so that the communication time can be masked by the longer duration of Gibbs sampling time. The trick here is that we have to make sure the target P_w processor is the same for all words in a bundle so that each time only one communication IO is required for fetching topic distributions for all words in a bundle.

For a P_d processor, we start bundling words according to their target P_w processors. For all words with the same target P_w processor, we first sort them in descending order of occurrence times and build a word list. We then iteratively pick a high frequency word from the head of the list and several low frequency words from the tail of the list and group them into a word bundle. After building word bundles, each time we will send a request to fetch topic distributions for all words in a bundle. For example, when learning topics from NIPS dataset consisting of 12-year NIPS papers, we combine $\{$*curve, collapse, compiler, conjunctive*, $\ldots\}$ as a bundle, in which *curve* is a high frequency word and the rest are low frequency words in this dataset.

3.4.3 *Building the Request Scheduler.* It is crucial to design an effective scheduler to determine the next word bundle to send requests for topic distributions during Gibbs sampling. We employ a simple pseudo-random scheduling scheme.

In this scheme, words in the vocabulary are stored in a circular queue. During Gibbs sampling, words are selected from this queue in a clockwise or counterclockwise order. Each P_d processor enters this circular queue with a different offset to avoid concurrent access to the same P_w processor. The starting point of each P_d process at each Gibbs sampling iteration is different. This randomness avoids forming the same bottlenecks from one iteration to another. Since circular scheduling is a static scheduling scheme, a bottleneck can still be formed at some P_w processors when multiple requests arrive at the same time. Consequently, some P_d processors may need to wait for a response before Gibbs sampling can start. We remedy this shortcoming by registering a deadline for each request, as described in Section 3.3.2. Requests on a P_w processor are processed according to their deadlines. A request will be discarded if its deadline has been missed. Due to the stochastic nature of Gibbs sampling, occasionally missing a round of Gibbs sampling does not affect overall performance. Our pseudo-random scheduling policy ensures the probability of same words being skipped repeatedly is negligibly low.

3.4.4 *Pipeline-based Gibbs Sampling.* Finally, we perform pipeline-based Gibbs sampling. As shown in Eq. (2), to compute and assign a new topic for a given word $x_{ij} = w$ in a document d_j, we have to obtain C_w^{word}, C^{topic} and C_j^{doc}. The topic distribution of document d_j is maintained by a P_d processor. While the up-to-date topic distribution C_w^{word} is maintained by a P_w processor, the global topic count C^{topic} should be collected over all P_w processors. Therefore, before assigning a new topic for a word w in a document, a P_d processor has to request C_w^{word} and C^{topic} from P_w processors. After fetching C_w^{word} and C^{topic}, the P_d processor computes and assigns new topics for occurrences of the word w. Then the P_d processor returns the updated topic distribution for the word w to the responsible P_w processor.

For a P_d processor *pd*, the pipeline scheme is performed according to the following steps:

(1) Fetch overall topic counts for Gibbs sampling.

(2) Select F word bundles and put them in the thread pool tp to fetch topic distributions for the words in each bundle. Once a request is responded by P_w processors, the returned topic distributions are put in a waiting queue Q_{pd}.

(3) For each word in Q_{pd}, pick its topic distribution to perform Gibbs sampling.

(4) After Gibbs sampling, put the updated topic distributions in the thread pool tp to send update requests to P_w processors.

(5) Select a new word bundle and put it in tp.

(6) If the update condition is met, fetch new overall topic counts.

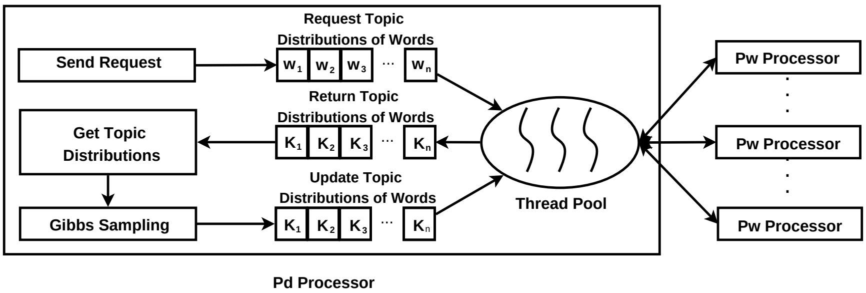

Fig. 6: The communication scheme of PLDA+.

(7) If the termination condition is not met, go to Step (3) to start Gibbs sampling for other words.

In Step (1), pd fetches the overall topic distributions C^{topic}. In this step, pd just sends the requests $fetch(pw, pd)$ to each P_w processor. The requests are returned with $C^{topic}_{|pw}, pw \in \{0, \ldots, |P_w| - 1\}$ from all P_w processors. Processor pd thus gets C^{topic} by summing overall topic counts from each P_w processor, $C^{topic} = \sum_{pw} C^{topic}_{|pw}$.

Since the thread pool tp can send requests and process the returned results in parallel, in Step (2) it puts a number of requests to fetch topic distributions simultaneously in case some requests are delayed. Since the requests are sent at the same time, they are assigned with the same deadline. Once a response is returned, it will start Gibbs sampling immediately. Here, we mention the number of pre-fetch requests as F. In PLDA+, F should be properly set to make sure the waiting queue Q_{pd} always has returned topic distributions of words waiting for Gibbs sampling. If not, it will stop to wait for the incoming member of Q_{pd}, which is a part of the communication time cost of PLDA+. To make best use of threads in the thread pool, F should be larger than the number of threads in the pool.

It is expensive for P_w processors to process the request for overall topic counts because the operation has to access the topic distributions for each word on each P_w processor. Fortunately, as indicated by the results of AD-LDA [Newman et al. 2009], topic assignments in Gibbs sampling are not sensitive to the values of the overall topic counts. We thus reduce the frequency of fetching overall topic counts to improve the efficiency of P_w processors. Therefore, in Step (6), we do not fetch overall topic counts frequently. In experiments, we will show that, by fetching new overall topic counts only after performing one pass of Gibbs sampling for all words, PLDA+ can obtain the same learning quality as LDA and PLDA.

The pipeline scheme is depicted in Fig. 6, where the process of fetching C^{topic} is not shown for simplicity.

3.4.5 *Fault Tolerance.* In PLDA+, we provide a fault-recovery solution similar to PLDA. We perform checkpointing only for $\boldsymbol{z}_{|pd}$ on P_d processors. This is because: (1) on the P_d side, $\boldsymbol{x}_{|pd}$ can be reloaded from dataset, and $C^{doc}_{|pd}$ can be recovered from $\boldsymbol{z}_{|pd}$; (2) on the P_w side, C^{word}_{pw} can also be recovered from $\boldsymbol{z}_{|pd}$. The recovery code is at the beginning of PLDA+: if there is a checkpoint on the disk, load it; otherwise perform random initialization.

3.5 Parameters and Complexity

In this section, we analyze parameters that may influence the performance of PLDA+. We also analyze the complexity of PLDA+ and compare it with PLDA.

3.5.1 *Parameters.* Given the total number of processors P, the first parameter is the proportion of the number of P_w processors to P_d processors, $\gamma = |P_w|/|P_d|$. The larger the value of γ, the more the average time for Gibbs sampling on P_d processors will increase as fewer processors are used to perform CPU-bound tasks. At the same time, the average time for communication will decrease since more processors serve as P_w to process requests. We have to balance the number of P_w and P_d processors to (1) minimize both computation and communication time, and (2) ensure that communication time is short enough to be masked by computation time. This parameter can be determined once we know the average time for Gibbs sampling and communication of the word-topic matrix. Suppose the total time for Gibbs sampling of the whole dataset is T_s, the communication time for transferring the topic distributions of all words from one processor to another processor is T_t. For P_d processors, the sampling time will be $T_s/|P_d|$. Suppose we transfer word topic distributions simultaneously to P_w processors, and thus transfer time will be $T_t/|P_w|$. To make sure the sampling process is able to overlap the fetching and updating process, we have to make sure

$$\frac{T_s}{|P_d|} > \frac{2T_t}{|P_w|}. \tag{5}$$

Suppose $T_s = W\bar{t_s}$ where $\bar{t}_s$ is the average sampling time for all instances of a word, and $T_t = W\bar{t}_f = W\bar{t}_u$, where $\bar{t}_f$ and $\bar{t}_u$ is the average fetching and update time for a word, we get

$$\gamma = \frac{|P_w|}{|P_d|} > \frac{\bar{t}_f + \bar{t}_u}{\bar{t}_s}, \tag{6}$$

where $\bar{t}_f$, $\bar{t}_u$ and $\bar{t}_s$ can be obtained by performing PLDA+ on a small dataset and then empirically set an appropriate γ value. Under the computing environment for our experiments, we empirically set $\gamma = 0.6$.

The second parameter is the number of threads in the thread pool R, which caps the number of parallel requests. Since the thread pool is used to prevent sampling from being blocked by busy P_w processors, R is determined by the network environment. R can be empirically tuned during Gibbs sampling. That is, when the waiting time for the prior iteration is long, the thread pool size is increased.

The third parameter is the number of requests F for pre-fetching the topic distributions before performing Gibbs sampling on P_d processors. This parameter depends on R, and in experiments we set $F = 2R$.

The last parameter is the maximum interval $inter_{max}$ for fetching the overall topic counts from all P_w processors during Gibbs sampling of P_d processors. This parameter influences the quality of PLDA+. In experiments, we can achieve LDA models with similar quality to PLDA and LDA by setting $inter_{max} = W$.

It should be noted that the optimal values of the parameters of PLDA+ are highly related to the distributed environment, including network bandwidth and processor speed.

3.5.2 *Complexity.* Table II summarizes the complexity of P_d processors and P_w processors in both time and space. For comparison, we also list the complexity of LDA and

Table II: Algorithm complexity. In this table, I is the iteration number of Gibbs sampling and c is a constant that converts bandwidth to flops.

Method	Time Complexity		Space Complexity
	Preprocessing	Gibbs sampling	
LDA	-	INK	$K(D+W)+N$
PLDA	$\frac{D}{\lvert P\rvert}$	$I\left(\frac{NK}{P}+cKW\log P\right)$	$\frac{(N+KD)}{P}+KW$
PLDA+,P_d	$\frac{D}{\lvert P_d\rvert}+cW\log W+\frac{WK}{\lvert P_w\rvert}$	$\frac{INK}{\lvert P_d\rvert}$	$\frac{(N+KD)}{\lvert P_d\rvert}$
PLDA+,P_w	-	-	$\frac{KW}{\lvert P_w\rvert}$

PLDA in this table. We assume $P = |P_w| + |P_d|$ when comparing PLDA+ with PLDA. In this table, I indicates the iteration number for Gibbs sampling, and c is a constant that converts bandwidth to flops.

The preprocessing of LDA distributes documents to P processors with time complexity $D/|P|$. Compared to PLDA, the preprocessing of PLDA+ requires three additional operations including (1) building an inverted document file for all documents on each P_d processor with time $O(D/|P_d|)$, (2) bundling words with time $O(W \log W)$ for fast sorting words according to their frequencies, and (3) sending topic counts from P_d processors to P_w processors to initialize the word-topic matrix on P_w with time $O(WK/|P_w|)$. In practice LDA is set to run with hundreds of iterations, and thus the preprocessing time for PLDA+ is insignificant compared to the training time.

Finally, let us consider the speedup efficiency of PLDA+. Suppose $\gamma = |P_w|/|P_d|$ for PLDA+, without considering preprocessing, the ideal achievable speedup is:

$$\text{speedup efficiency} = \frac{S/P}{S/|P_d|} = \frac{|P_d|}{P} = \frac{1}{1+\gamma}, \tag{7}$$

where S denotes the running time for LDA on a single processor, S/P is the ideal time cost using P processors, and $S/|P_d|$ is the ideal time achieved by PLDA+ with communication completely masked by Gibbs sampling.

4. EXPERIMENTAL RESULTS

We compared the performance of PLDA+ with PLDA (AD-LDA based) through empirical study. Our study focused on comparing both training quality and scalability. Since the speedups of AS-LDA are just “competitive” to those reported for AD-LDA as shown in [Asuncion et al. 2008; 2010], we chose not to compare with AS-LDA.

4.1 Datasets and Experiment Environment

We used the three datasets shown in Table III. The NIPS dataset consists of scientific articles from NIPS conferences. The NIPS dataset is relatively small, and we used it to investigate the influence of missed deadlines on training quality. Two Wikipedia datasets were collected from English Wikipedia articles using the March 2008 snapshot from `en.wikipedia.org`. By setting the size of the vocabulary to $20,000$ and $200,000$, respectively, the two Wikipedia datasets are named Wiki-20T and Wiki-200T. Compared to Wiki-20T, more infrequent words are added in vocabulary in Wiki-200T. However, even for those words ranked around $200,000$, they have occurred in more than 24 articles in Wikipedia, which is sufficient to learn and infer their topics using LDA. These two large

Table III: Detailed information of data sets.

	NIPS	Wiki-20T	Wiki-200T
D_{train}	1,540	2,122,618	2,122,618
W	11,909	20,000	200,000
N	1,260,732	447,004,756	486,904,674
D_{test}	200	-	-

datasets were used for testing the scalability of PLDA+. In experiments, we implemented PLDA+ using a synchronous remote procedure call (RPC) mechanism. The experiments were run on a distributed computing environment with $2,048$ processors, each with a 2GHz CPU, 3GB of memory, and a disk allocation of 100GB.

4.2 Impact of Missed Deadlines

Similar to [Newman et al. 2007], we use *test set perplexity* to measure the quality of LDA models learned by various distributed methods of LDA. Perplexity is a common way of evaluating language models in natural language processing, computed as:

$$Perp(\boldsymbol{x}^{\text{test}}) = \exp\big(-\frac{1}{N^{\text{test}}}\log p(\boldsymbol{x}^{\text{test}})\big), \tag{8}$$

where $\boldsymbol{x}^{\text{test}}$ denotes the test set, and N^{test} is the size of the test set. A lower perplexity value indicates a better quality. For every test document in the test set, we randomly designated half the words for fold-in, and the remaining words were used for testing. The document mixture $\boldsymbol{\theta}_j$ was learned using the fold-in part, and the log probability of the test words was computed using this mixture. This arrangement ensures that the test words were not used in estimating model parameters. The perplexity computation follows the standard method in [Griffiths and Steyvers 2004], which averages over multiple chains when making predictions using LDA models learned by Gibbs sampling. Using perplexity on the NIPS dataset, we find the quality and convergence rate of PLDA+ are comparable to single-processor LDA and PLDA. Since the conclusion is straightforward and similar to [Newman et al. 2007], we do not present the evaluation results on perplexity in detail.

As described in Section 3.4.3, PLDA+ discards a request when its deadline is missed. Here we investigate the impact of missed deadlines on training quality using the NIPS dataset. We define *missing ratio* δ as the average number of missed requests divided by the total number of requests, which ranges $[0.0, 1.0)$. By randomly dropping δ requests in each iteration, we simulated discarding different amounts of requests in each iteration. We compared the quality of learned topic models under different δ values. In experiments we set $P = 50$. Fig. 7 shows the perplexities with different δ values versus the number of sampling iterations when $K = 10$. When the missing ratio is less than 60%, the perplexities remain reasonable. At interaction 400, the perplexities of δ's between 20% and 60% are about the same, whereas no deadline misses can achieve a 2% better perplexity. Qualitatively, a 2% perplexity drop does not show any discernible degradation in training results. Fig. 8 shows the perplexities of converged topic models with various numbers of topics versus different δ settings, at the end of iteration 400. A larger K setting suffers from more severe perplexity degradation. Nevertheless, $\delta = 60\%$ seems to be a limiting threshold that PLDA+ can endure. In reality, our experiments indicate that the missing ratio is typically lower than 1%, far from the limiting threshold. Though the missing ratio depends highly on the workload and the computation environment, the result of this

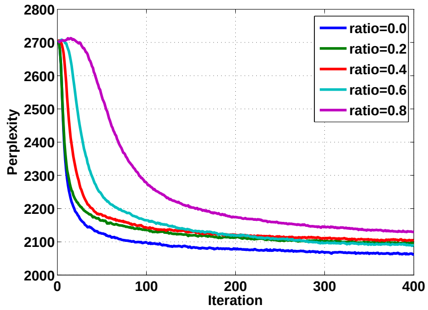

Fig. 7: Perplexity versus the number of iterations when missing ratio is 0.0, 0.2, 0.4, 0.6 and 0.8.

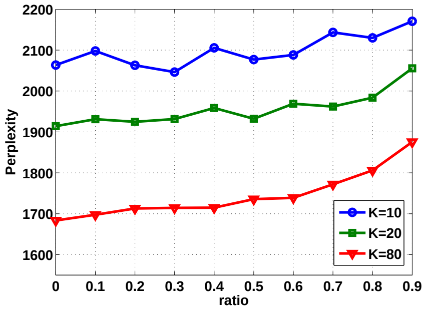

Fig. 8: Perplexity with various numbers of topics versus missing ratio.

experiment is encouraging that PLDA+ can operate well even when δ is high.

4.3 Speedups and Scalability

The primary motivation for developing distributed algorithms for LDA is to achieve a good speedup. In this section, we report the speedup of PLDA+ compared to PLDA. We used Wiki-20T and Wiki-200T for speedup experiments. By setting the number of topics $K = 1,000$, we ran PLDA+ and PLDA on Wiki-20T using $P = 64, 128, 256, 512$ and $1,024$ processors, and on Wiki-200T using $P = 64, 128, 256, 512, 1,024$ and $2,048$ processors. Note that for PLDA+, $P = P_w + P_d$, and the ratio of $|P_w|/|P_d|$ was empirically set to $\gamma = 0.6$ according to the unit sampling time and transfer time. The number of threads in a thread pool was set as $R = 50$, which was determined based on the experiment results. As analyzed in Section 3.5.2, the ideal speedup efficiency of PLDA+ is $\frac{1}{1+\gamma} = 0.625$.

Fig. 9 compares speedup performance on Wiki-20T. The speedup was computed relative to the time per iteration when using $P = 64$ processors, because it was impossible to run the algorithms on a smaller number of processors due to memory limitations. We assumed the speedup on $P = 64$ to be 64, and then extrapolated on that basis. From the figure, we observe that when P increases, PLDA+ simply achieves much better speedup than PLDA, thanks to the much reduced communication bottleneck of PLDA+. Fig. 10 compares the ratio of communication time over computation time on Wiki-20T. When $P = 1,024$, the communication time of PLDA is 13.38 seconds, which is about the same as its computation time, much longer than that of PLDA+'s 3.68 seconds.

From the results, we conclude that: (1) When the number of processors grows large enough (e.g., $P = 512$), PLDA+ begins to achieve better speedup than PLDA; (2) In fact, if we take the waiting time for synchronization in PLDA into consideration, the speedup of PLDA could have been even worse. For example, in a busy distributed computing environment, when $P = 128$, PLDA may take about 70 seconds for communication in which only about 10 seconds are used for transmitting word-topic matrices and most of the time is used to wait for computation to complete.

On the larger Wiki-200T dataset, as shown in Fig. 11, the speedup of PLDA starts to flatten out at $P = 512$, whereas PLDA+ continues to gain in speed[1]. For this dataset,

[1] For PLDA+, the parameter of pre-fetch number and thread pool size was set to $F = 100$ and $R = 50$. With $W = 200,000$ and $K = 1,000$, the matrix is 1.6GBytes, which is large for communication.

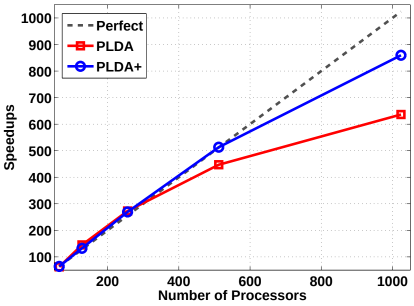

Fig. 9: Parallel speedup results for 64 to 1, 024 processors on Wiki-20T.

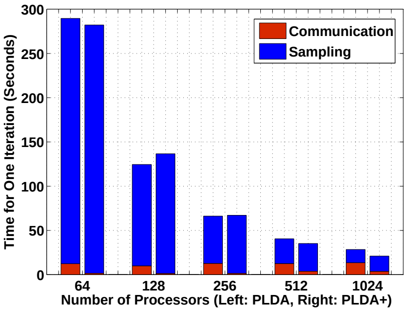

Fig. 10: Communication and sampling time for 64 to 1, 024 processors on Wiki-20T.

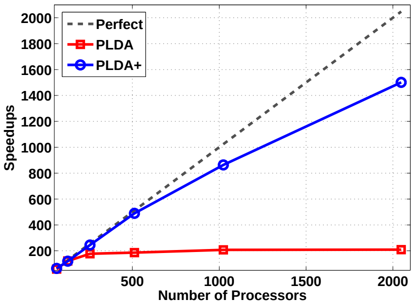

Fig. 11: Parallel speedup results for 64 to 2, 048 processors on Wiki-200T.

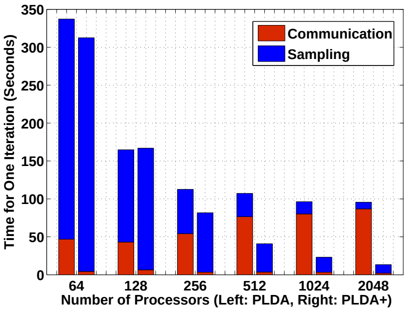

Fig. 12: Communication and sampling time for 64 to 2, 048 processors on Wiki-200T.

we also list the sampling and communication time ratio of PLDA and PLDA+ in Fig. 12. PLDA+ keeps the communication time to consistently low values from $P = 64$ to $P = 2,048$. When $P = 2,048$, PLDA+ took only about 20 minutes to finish 100 iterations while PLDA took about 160 minutes. Though eventually Amdahl's Law would kick in to cap speedup, it is evident that the reduced overhead of PLDA+ permits it to achieve much better speedup for training on large-scale datasets using more processors.

The above comparison did not take preprocessing into consideration because the preprocessing time of PLDA+ is insignificant compared to the training time as analyzed in Section 3.5.2. For example, the preprocessing time for the experiment setting of $P = 2,048$ on Wiki-200T is 35 seconds. For training, hundreds of iterations are required, with each iteration taking about 13 seconds.

5. CONCLUSION

In this paper, we presented PLDA+, which employs data placement, pipeline processing, word bundling, and priority-based scheduling strategies to substantially reduce intercomputer communication time. Extensive experiments on large-scale datasets demonstrated that PLDA+ can achieve much better speedup than previous attempts on a distributed environment.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. We also thank Matt Stanton, Hongjie Bai, Wen-Yen Chen and Yi Wang for their pioneering work, and thank Xiance Si, Hongji Bao, Tom Chao Zhou, Zhiyu Wang for helpful discussions. This work is supported by Google-Tsinghua joint research grant.

REFERENCES

ASUNCION, A., SMYTH, P., AND WELLING, M. 2008. Asynchronous distributed learning of topic models. In *Proceedings of NIPS*. 81–88.

ASUNCION, A., SMYTH, P., AND WELLING, M. 2010. Asynchronous distributed estimation of topic models for document analysis. *Statistical Methodology*.

BERENBRINK, P., FRIEDETZKY, T., HU, Z., AND MARTIN, R. 2008. On weighted balls-into-bins games. *Theoretical Computer Science 409,* 3, 511–520.

BLEI, D. M., NG, A. Y., AND JORDAN, M. I. 2003. Latent dirichlet allocation. *Journal of Machine Learning Research 3,* 993–1022.

BLINN, J. 1991. A trip down the graphics pipeline: Line clipping. *IEEE Computer Graphics and Applications 11,* 1, 98–105.

CHEMUDUGUNTA, C., SMYTH, P., AND STEYVERS, M. 2007. Modeling general and specific aspects of documents with a probabilistic topic model. In *Proceedings of NIPS*. 241–248.

CHEN, W., CHU, J., LUAN, J., BAI, H., WANG, Y., AND CHANG, E. 2009. Collaborative filtering for orkut communities: discovery of user latent behavior. In *Proceedings of WWW*. 681–690.

CHU, C.-T., KIM, S. K., LIN, Y.-A., YU, Y., BRADSKI, G., NG, A. Y., AND OLUKOTUN, K. 2006. Mapreduce for machine learning on multicore. In *Proceedings of NIPS*.

DEAN, J. AND GHEMAWAT, S. 2004. Mapreduce: Simplified data processing on large clusters. In *Proceedings of OSDI*. 137–150.

GOMES, R., WELLING, M., AND PERONA, P. 2008. Memory bounded inference in topic models. In *Proceedings of ICML*. 344–351.

GRAHAM, S., SNIR, M., AND PATTERSON, C. 2005. *Getting up to speed: The future of supercomputing*. National Academies Press.

GRIFFITHS, T. AND STEYVERS, M. 2004. Finding scientific topics. *Proceedings of the National Academy of Sciences of the United States of America 101,* 90001, 5228–5235.

LI, W. AND MCCALLUM, A. 2006. Pachinko allocation: DAG-structured mixture models of topic correlations. In *Proceedings of ICML*.

MIMNO, D. M. AND MCCALLUM, A. 2007. Organizing the OCA: learning faceted subjects from a library of digital books. In *Proceedings of ACM/IEEE Joint Conference on Digital Libraries*. 376–385.

NEWMAN, D., ASUNCION, A., SMYTH, P., AND WELLING, M. 2007. Distributed inference for latent dirichlet allocation. In *Proceedings of NIPS*. 1081–1088.

NEWMAN, D., ASUNCION, A., SMYTH, P., AND WELLING, M. 2009. Distributed algorithms for topic models. *Journal of Machine Learning Research 10,* 1801–1828.

PORTEOUS, I., NEWMAN, D., IHLER, A., ASUNCION, A., SMYTH, P., AND WELLING, M. 2008. Fast collapsed gibbs sampling for latent dirichlet allocation. In *Proceedings of KDD*. 569–577.

ROSEN-ZVI, M., CHEMUDUGUNTA, C., GRIFFITHS, T., SMYTH, P., AND STEYVERS, M. 2010. Learning author-topic models from text corpora. *ACM Transactions on Information Systems 28,* 1, 1–38.

SHEN, J. P. AND LIPASTI, M. H. 2005. *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill Higher Education.

WANG, Y., BAI, H., STANTON, M., CHEN, W., AND CHANG, E. 2009. PLDA: Parallel latent dirichlet allocation for large-scale applications. In *Algorithmic Aspects in Information and Management*. 301–314.

YAN, F., XU, N., AND QI, Y. 2009. Parallel inference for latent dirichlet allocation on graphics processing units. In *Proceedings of NIPS*. 2134–2142.